VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Server pro podporu výuky teoretické informatiky

Supporting Server for Theoretical Computer Science Teaching

2017 Bc. Michael Janošík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michael Janošík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Server pro podporu výuky teoretické informatiky
Supporting Server for Theoretical Computer Science Teaching

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem diplomové práce je vytvořit server, na kterém si studenti budou moci spustit výpočet vybraných algoritmů z oblasti teoretické informatiky na jimi zadaných i ukázkových vstupech a na zobrazeném průběhu výpočtu lépe pochopit princip fungování těchto algoritmů. Tato práce se konkrétně zaměří na algoritmy z oblasti konečných automatů.

1. Naprogramujte řešení problémů z oblasti konečných automatů. Konkrétně simulování běhu (N)KA na zadaném slově, redukce KA, konstrukce automatů pro průnik, sjednocení, doplněk a zrcadlový obraz jazyka.
2. Cílem není nejefektivnější implementace algoritmů, ale taková, která umožní zobrazovat postupně jednotlivé kroky tak, aby uživatel mohl pochopit princip jejich fungování.
3. Vytvořte výukový server, kde bude možné zadat vstupní data pro naprogamované algoritmy a pro tato data si nechat zobrazit postup výpočtu.
4. Vytvořte i ukázkové vstupy pro jednotlivé algoritmy, aby postup výpočtu bylo možné zobrazit i bez zadávání vstupu uživatelem.

Seznam doporučené odborné literatury:

[1] P. Jančar: Teoretická informatika (výukový text), FEI VŠB - TUO, 2007.

Další literatura dle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017 .....

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce panu Ing. Martinu Kotovi, Ph.D., jehož připomínky a celková odborná pomoc přispěly při vypracování této práce.

Abstrakt

Cílem této diplomové práce je vytvoření serveru pro podporu výuky předmětu „Teoretické informatiky“, vyučovaném na VŠB-TUO. Přesněji se pak tato práce zaměřuje pouze na algoritmy z oblasti konečných automatů. Předmět „Toeretická informatika“ navazuje na úvodní předmět z bakalářského studia, kde se také tyto algoritmy vyučovaly. Práce se zaměřuje na vytvoření automatu uživatelem a výpočtu algoritmů po jednotlivých krocích. Celá práce je pak ve výsledku vyřešena webovou aplikací, umožňující vytvořit automat pomocí grafu, nebo přechodovou tabulkou. Tyto vstupy jsou zadávány přímo do webového prohlížeče. Jednotlivé kroky algoritmů se v aplikaci uživatel učí pomocí grafického znázornění u grafu automatu a textovými popisy. Výsledek také obsahuje uložené příklady, které si uživatel může jednoduše zobrazit, a funkce pro uložení do souboru a načtení automatu ze souboru.

Klíčová slova: Teoretická informatika, algoritmus, konečný automat, ASP.NET, webová aplikace

Abstract

The aim of this thesis is to create a server to support the teaching of the subject „Theoretical Computer Science“ taught at VŠB-TUO. More specifically, this work focuses only on algorithms from area of finite-state automata. Subject „Theoretical Computer Science“ follows the introductory course from a bachelor's degree, where are these algorithms also taught. Diploma thesis in the resulting web application allows users to create an automaton using a graph representation or transition table. The thesis focuses on the creation of automaton by user and the computation of algorithms by individual steps. The result is a web application that allows you to create a finite-state automaton by a graph or a transition table. These inputs are entered directly into the web browser. The algorithm's individual steps are taught in the application by the graphical representation of an automaton graph and textual descriptions. The result also contains stored examples, which user can easily load, and functions for saving an automaton into a file and loading machine from a file.

Key Words: Theoretical Computer Science, algorithm, finite state machine, ASP.NET, web application

Obsah

Seznam použitých zkratek a symbolů

KA	–	Konečný automat
DKA	–	Deterministický konečný automat
NKA	–	Nedeterministkický konečný automat
ASP	–	Active Server Pages
HTML	–	Hyper Text Markup Language
XML	–	eXtensible Markup Language
AJAX	–	Asynchronous JavaScript and XML
IIS	–	Internet Information Services
CSS	–	Cascading Style Sheets
MS	–	Microsoft
API	–	Application Programming Interface
MVC	–	Model-view-controller
VB	–	Visual Basic
UI	–	User interface
REST	–	Representational State Transfer
URL	–	Uniform Resource Locator
JSON	–	JavaScript Object Notation
UML	–	Unified Modeling Language
MSDN	–	Microsoft Developer Network

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Teoretická informatika studuje typy problémů, objevující se v informatice. K analýze a zkoumání těchto problémů využívá matematické prostředky. Charakteristické pro teoretickou informatiku je její rigorózní matematický přístup, kde je důraz kladen především na důkazy a přesné formální definice pojmů, se kterými se pracuje. Hlavní pojem, který teoretická informatika zkoumá, je algoritmus. Pojmem algoritmus se rozumí přesný návod, či postup, kterým lze vyřešit daný typ úlohy, přičemž je tento postup většinou realizován pomocí počítačového programu. Tím se teoretická informatika odlišuje od jiných oblastí matematiky.

Cílem této diplomové práce je vytvoření webového serveru, sloužícího studentům k snadnějšímu pochopení algoritmů využívajících konečné automaty, právě v oboru teoretické informatiky. Toho bude docíleno pomocí jednotlivých komponent aplikace, které budou umět snadné vytvoření jednotlivých automatů a následné grafické nebo textové znázornění postupu určitého algoritmu. Důraz je kladen na jednoduchost vkládání vstupů a znázornění postupů pro rychlejší pochopení daného algoritmu.

Celá diplomová práce je rozdělena do dvou částí. V první části, jsou popsány základní pojmy a postupy, které jsou využívány algoritmy konečných automatů (ke každému algoritmu uvedu i menší příklad). Prostor je zde hlavně věnován způsobům zápisu konečného automatu a samotným algoritmům, použitých v této diplomové práci. Ve druhé části je popsán návrh a implementace výsledného programového řešení. Prvně jsou zodpovězeny základní otázky k funkčnosti webové aplikace a poté je popsáno uživatelské rozhraní.

Hlavní kapitolou poté bude Implementace celého serveru, kde se budu snažit popsat jednotlivé vstupy a výstupy aplikace a následné implementace jednotlivých algoritmů, tak aby se dali co nejjednodušeji pochopit studentem (uživatelem). Po této kapitole následuje část věnovaná testování a nasazení do provozu a následné zhodnocení výsledku práce.

2 Konečné automaty

V této kapitole jsou obsaženy základní pojmy konečných automatů a algoritmů, kterými se tato práce zabývá.

2.1 Definice

Konečný automat popisuje jednoduchý počítač, který může být v jednom z několika stavů. Tyto stavy jsou děleny na dva druhy a to přijímající a nepřijímající. Automat mezi stavy přechází na základě vstupních symbolů. Jméno Konečný vyplývá právě ze stavů, jichž je konečně mnoho. Automat kromě stavů, ve kterých se může při přečtení dalšího znaku nacházet, nemá žádnou další paměť. Celý automat je pak dán konečnou neprázdnou množinou stavů, konečnou neprázdnou množinou vstupních symbolů nazývanou abeceda, přechodovou tabulkou popisující pravidla přechodů mezi stavy, počátečními stavy a nakonec přijímajícími stavy.

Definice 1 [1]
Formálně se pak automat zapisuje jako uspořádaná pětice $A = (Q, \Sigma, \sigma, q_0, F)$*, kde:*

- Q *je konečná neprázdná množina stavů*
- Σ *je konečná neprázdná množina zvaná (vstupní) abeceda*
- $\sigma : Q \times \Sigma \to Q$ *je přechodová funkce*
- $q_0 \in Q$ *je počáteční (iniciální) stav*
- $F \subseteq Q$ *je množina přijímajících (koncových) stavů*

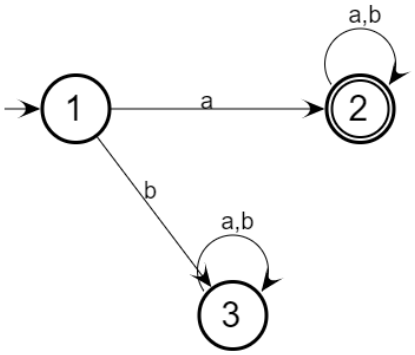
Přechodová funkce se zapisuje jako $\delta(q_1, a) = q_2$*, kde* $q_1 \in Q$*,* $a \in \Sigma$ *a výsledek funkce* $q_2 \in Q$*.*

2.2 Zápis automatu

Nejčastějším korektním zápisem automatu je diagram, který se také nazývá stavový diagram, nebo graf automatu. Nejjednodušší pochopení tohoto zápisu je přímo pomocí příkladu.

Na příkladu vyobrazeném na Obrázku 1 můžeme vidět automat $A = (Q, \Sigma, \sigma, q_0, F)$, s množinou stavů $Q = \{1, 2, 3\}$, dále abecedou $\Sigma = \{a, b\}$, jedním počátečním stavem $q_0 = 1$, množinou přijímajících stavů $F = \{2\}$ a pro přechodovou funkci platí například $\delta(1, a) = 2$, $\delta(2, a) = \delta(2, b) = 2$, atd.. Z toho jednoznačně vyplývá:

- vrcholy jsou stavy automatu (prvky množiny Q)
- příchozí šipkou (vedenou odnikud) je znázorněn počáteční stav q_0
- koncové stavy (zde jenom stav 2) jsou označeny dvojitým kroužkem



Obrázek 1: Příklad zápisu automatu grafem

- přechodové funkce jsou hrany grafu, kde znak přechodu je označen nad hranou a jedna hrana může mít více znaků pro přehlednější zápis

Dále se můžeme setkat například s automaty se zápisem pomocí grafu, který má označené přijímající stavy šipkou vedenou ze stavu do volného prostoru, ale v této práci budu vyžívat zápisu popsaného v předchozím příkladu (Obrázek 1).

Další zápis automatu, používaný v této práci a ve výsledné aplikaci je pomocí tzv. přechodové tabulky. Rozumí se tím tabulka s řádky označenými stavy automatu a sloupci označenými symboly abecedy, ve které políčko na řádku q_1 a sloupci označeným znakem a udává stav q_2 (tzn. Přechodová funkce $\sigma(q_1, a) = q_2$). Počáteční stavy jsou značeny šipkou $\rightarrow$, přijímající zase $\leftarrow$, kde tyto šipky jsou přímo na začátku řádku u stavu (pokud stav je počáteční a také přijímající, šipky se spojí v $\leftrightarrow$).

	a	b
$\rightarrow$ 1	2	3
$\leftarrow$ 2	2	2
3	3	3

Tabulka 1: Automat z Obrázku 1 v zápise pomocí přechodové tabulky

2.3 Deterministický konečný automat

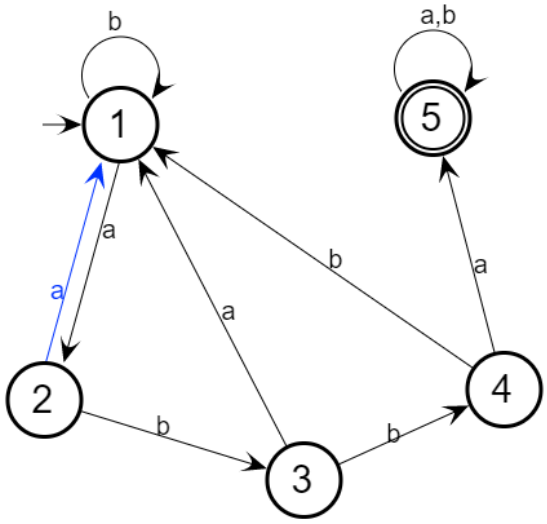
Definice toho automatu byla popsána v minulé části. Zde jenom popíšu hlavní charakteristiky.

Deterministický konečný automat (DKA) má vždy jen jeden počáteční stav, ale přijímajících stavů, může být jakékoliv množství. Dále je hlavní, aby přechodová funce vracela právě jenom jeden stav pro každý znak v abecedě automatu.

2.4 Nedeterministický konečný automat

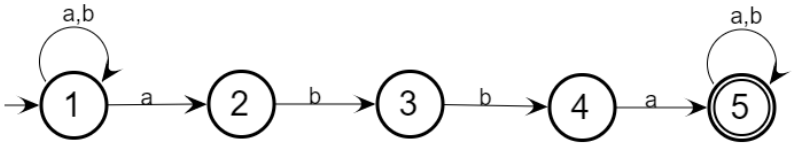
Pro pochopení nedeterministického konečného automatu (NKA) a celého pojmu nedeterministického výpočtu zde zvolím postup vysvětlení přímo na příkladu.

Mějme automat rozpoznávající jazyk $\{w \in \{a,b\}^* | w$ obsahuje podslovo $abba\}$. Pokud by se nad tímto jazykem vytvořil deterministický automat, výsledný graf automatu by vypadal jako na Obrázku 2.



Obrázek 2: Příklad DKA grafem

Zde vidíme, že automat není zas tak triviální pro takhle jednoduchý jazyk, jako je obsažení řetězce ve slově. Proto existují automaty tzv. nedeterministické, které umožňují jak lehčí návrh celého automatu, tak také zjednodušení operací s automaty jako je např. zřetězení dvou jazyků, nebo sjednocení automatů představující tyhle jazyky. Tyto automaty jsou pak dány přechodovými funkcemi, které nevrací jeden stav ale rovnou množinu stavů, kde pro daný znak v daném stavu může být tato množina prázdná.



Obrázek 3: Příklad z Obrázku 2 pomocí NKA

Automat se nám poté zkrátil o pár přechodových funkcí, přitom funkcionalitu neztratil (přijímá stejný jazyk jak automat na Obrázku 2). NKA slovo přijímá, když existuje alespoň jedna

možnost jak dojít z počátečního do přijímajícího stavu. Musím ještě zmínit, že nedeterministický automat může mít více počátečních stavů a také přechodová funkce může dostat jako druhý argument kromě písmena abecedy i speciální znak epsilon („ε“). Epsilon znak přesněji znamená, že přechod automaticky, při dosažení stavu odkud vede, výpočet rozděluje i na stav, který z tohoto přechodu vyplývá.

Definice 2 [1]

Nedeterministický konečný automat je dán uspořádanou pěticí $A = (Q, \Sigma, \sigma, I, F)$*, kde:*

- Q *je konečná neprázdná množina stavů*
- Σ *je konečná neprázdná abeceda*
- $\sigma : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ *je přechodová funkce*
- $I \in Q$ *je množina počátečních (iniciálních) stavů*
- $F \in Q$ *je množina přijímajících (koncových) stavů*

Rozdíl jak vidíme je především v přechodových funkcích, kde teď $\sigma(q, a)$ nepředstavuje jeden stav, ale množinu stavů (z nich může být jakýkoli vybrán jako následující). Další rozdíl je v tom, že počáteční stav není jenom jeden stav, ale skládá se z množiny stavů.

Samozřejmě můžeme i NKA zadávat přechodovou tabulkou. V políčkách pak nebude jeden stav, ale množiny stavů, kde obvykle vynecháváme složené závorky. Poté, pokud množina stavů je prázdná, se často místo symbolu prázdné množiny používá pomlčka. Automat z posledního příkladu pak pomocí tabulky vypadá takto:

	a	b
$\to 2$	-	3
3	-	4
4	5	-
$\leftarrow 5$	5	5

Tabulka 2: Zápis NKA pomocí přechodové tabulky

2.5 Automat jako rozpoznávač regulárních jazyků (simulování běhu automatu)

Zde popíšu hlavní činnost konečného automatu a to přijímání zadaného slova automatem. Přesněji pak zadané slovo je součástí regulárního jazyka, který automat rozpoznává. Kde definice regulárního jazyka právě vyplývá z toho, že daný jazyk je regulární právě tehdy, když je akceptovaný nějakým konečným automatem.

Ze začátku si představme, že nevidíme automat (tedy nevidíme graf ani přechodovou tabulku, prostě nic co by popisovalo přechodovou funkci σ), ale jsme v pozici vnějšího pozorovatele. Vidíme jen řídící jednotku (nevidíme vnitřek této jednotky), která je čtecí hlavou spojena s páskou, na níž je zapsané vstupní slovo (znaky jsou napsané zleva doprava v jednotlivých buňkách pásky). Na začátku je tato jednotka v počátečním stavu a čtecí hlava je na začátku slova (nejlevější políčko na pásce). Poté výpočet automatu probíhá po krocích, kdy v každém kroku je přečten znak (hlava se po přečtení posune o jedno políčko doprava) a řídící jednotka se nastaví do stavu určeného aktuálním stavem a přečteným znakem (tzn. podle přechodové funkce). Poté už záleží na posledním přečteném znaku, kde automat simulaci končí a pokud končí v přijímajícím stavu, automat slovo přijal, a tudíž je slovo součástí jazyka, který automat rozpoznává.

Definice 3 [1]

Mějme konečný automat $A = (Q, \Sigma, \sigma, q_0, F)$.

Slovo $w \in \Sigma^*$ *je přijímáno automatem A, jestliže* $q_0 \xrightarrow{w} F$. *Jazykem rozpoznávaným (přijímaným) automatem A rozumíme jazyk* $L(A) = \{w \in \Sigma^* |$ *slovo w je přijímáno A*$\} = \{w \in \Sigma^* | q_0 \xrightarrow{w} F\}$.

Taky se zde musím zmínit o rozdílu průběhu výpočtu slova na nedeterministickém automatu oproti deterministického. Hlavní rozdíl je v tom, že u nedeterministického automatu po přečtení znaku samotná přechodová funkce vrací množinu stavů. Toto nakonec vede k „nedeterministickému rozhodování“, který stav z množiny využije (proto také název automatu nedeterministický). Nakonec dostaneme pro jedno a totéž slovo w více možných výpočtů. Pro úplnost vysvětlení musím zmínit fakt, že pokud nastane situace, kdy se dostaneme čtením slova do stavu, odkud přechodová funkce, znakem dalším v pořadí čtení, vrací prázdnou množinu, výpočet slovo automaticky nepřijímá. Pokud existuje výpočet v NKA, který došel do přijímajícího stavu na konci čtení, automat slovo přijímá. Naopak, pro deterministický automat vždy existuje jenom jeden výpočet na zadaném slově.

Př. Výpočet slova *aabaa* na automatu popsaného přechodovou tabulkou:

	a	b
$\leftrightarrow 1$	2	2
2	1	1

Tabulka 3: Přechodová tabulka Automatu pro příklad simulování běhu automatu

Na první pohled vidíme automat, který by se dal popsat jako $L = \{w \in \{a, b\}^* | w$ má sudou délku $\}$. Čtecí hlava je na začátku slova (tudíž na znaku „a“) a automat je v počátečním stavu označeném 1. Poté probíhá první čtení znaku čtecí hlavou a podle přechodové funkce $\sigma(1, a)$, která automat převede do stavu 2 se čtecí hlava posune o jedno políčko doprava a proces se opakuje přečtením dalšího znaku čtecí hlavou. Tento proces se opakuje až do posledního znaku,

který udává, v jakém konečném stavu se automat nachází (buď ve stavu přijímajícím, nebo nepřijímajícím). V našem případě slovo *aabaa* skončí ve stavu 2, který není přijímající a slovo tudíž do jazyka nepatří.

2.6 Převod nedeterministického automatu na deterministický

Bylo už vysvětleno, jak lze sestrojit NKA a definovat přijímání slov a jazyků pomocí tohoto automatu. Pro každý nedeterministický automat existuje minimálně jeden deterministický přijímající stejný jazyk (tzn. síla NKA je ekvivalentní síle DKA). Toto se dá dokázat pomocí algoritmu, který dokáže sestrojit ke každému NKA ekvivalentní (deterministický) konečný automat A' (tedy $L(A) = L(A')$).

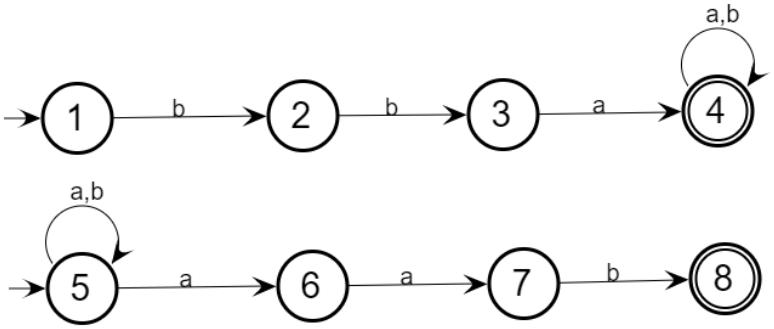
Algoritmus pak čerpá z toho, že NKA může být zároveň ve více stavech, takže se můžeme dostat do situace, kdy je náš NKA ve stavech$\{q_0, q_1\}$. Přečtením dalšího znaku, například 0, se můžeme dostat do nové množiny $\{q_1, q_3, q_4\}$. V deterministické verzi toho automatu se to projeví tím, že tyto množiny stavů budou mít jeden stav pojmenovaný stejně jako tyto množiny (stav $\{q_0, q_1\}$ a stav $\{q_1, q_3, q_4\}$) s přechodovou funkcí pro znak 0. Formálně se tak dostaneme z jednoho stavu do druhého, ale díky pojmenování budeme vědět, že v nedeterministickém automatu bychom se ocitli ve více stavech najednou. Pokud neexistuje přechodová funkce v celé množině stavů, ve které se zrovna nacházíme, víme, že NKA slovo nepřijímá, proto vytvoříme v DKA nový stav nepřijímající, do kterého povedeme přechod daným znakem a zde výpočet zacyklíme přechodem pro všechny znaky abecedy vycházející z tohoto stavu a mířícího do tohoto stavu.

Definice 4 [1]
Máme NKA $A = (Q, \Sigma, \sigma, I, F)$ *a k němu právě DKA* $A' = (Q', \Sigma, \sigma', I, F')$ *kde:*

- $Q' = P(Q)$ *je množina všech podmnožin stavů* Q
- *množina* $I \in Q'$ *je počátečním stavem*
- $F' = \{K \in Q' | K \cap F \neq \emptyset\}$ *(F' tedy obsahuje všechny podmnožiny množiny Q, které obsahují některý stav z F)*
- *Přechodová funkce* $\sigma' : Q' \times \Sigma \to Q'$ *každé podmnožiny původních stavů* $K \subseteq Q$ *a písmenu* $a \in \Sigma$ *přiřadí podmnožinu* $\{q \in Q | \exists q' \in K : q' \xrightarrow{a} q\}$

Výsledný automat pak jde většinou v mnoha případech zmenšit následnou redukcí, ke které se zde dostanu v následující podkapitole.

Př. Mějme nedeterministický konečný automat pospaný následujícím grafem



Obrázek 4: Automat pro příklad převodu NKA na DKA

K sestrojení si pomůžeme novou přechodovou tabulkou, která bude mít na začátku jenom jeden stav, sestavující se ze stavu počátečních $\{1, 5\}$, který bude také počáteční. Jestli je stav přijímající rozhoduje, zda některý ze stavů přijímající je (v tomto případě stav $\{1, 5\}$ přijímající není). Do tabulky můžeme rovnou zapsat výsledky přechodové funkce pro jednotlivé znaky abecedy (tzn. množiny stavů do kterých se dostaneme přečtením daného znaku). Tabulka bude poté vypadat takto:

	a	b
$\leftrightarrow$ {1, 5}	{5, 6}	{2, 5}

Tabulka 4: Tabulka v prním kroku převodu NKA na DKA

V dalším kroku se podíváme, jestli množiny, do kterých jsme s přechodovou funkcí došli, nemáme už v tabulce zapsané. Pokud nemáme, píšeme na další řádek tuto množinu jako nový stav a znovu sestrojíme k tomuto stavu přechodové funkce, jako v minulém kroku (v tomto případě množiny {5, 6} a {2, 5} v tabulce nejsou a musí se do tabulky zapsat).

	a	b
$\leftrightarrow$ {1, 5}	{5, 6}	{2, 5}
{5, 6}	{5, 6, 7}	{5}
{2, 5}	{5, 6}	{3, 5}

Tabulka 5: Tabulka v druhém kroku převodu NKA na DKA

Dále postupujeme pořád stejně do té doby, než automat bude deterministický (z každého popsaného stavu každým znakem přechodová funkce vrací jeden stav). Výsledný automat z toho příkladu pak vypadá následovně:

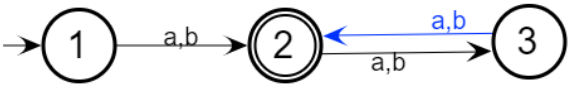
Zde můžeme také vidět, že přijímající stavy jsou ty, které obsahují stav 4, nebo 8, z předešlého NKA.

	a	b
↔ {1, 5}	{5, 6}	{2, 5}
{5, 6}	{5, 6, 7}	{5}
{2, 5}	{5, 6}	{3, 5}
{5, 6, 7}	{5, 6, 7}	{5, 8}
{5}	{5, 6}	{5}
{3, 5}	{4, 5, 6}	{5}
←{5, 8}	{5, 6}	{5}
←{4, 5, 6}	{4, 5, 6, 7}	{4, 5}
←{4, 5, 6, 7}	{4, 5, 6, 7}	{4, 5, 8}
←{4, 5}	{4, 5, 6}	{4, 5}
←{4, 5, 8}	{4, 5, 6}	{4, 5}

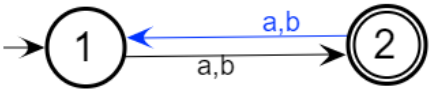
Tabulka 6: Tabulka v posledním kroku převodu NKA na DKA

2.7 Redukce automatu

Ke každému automatu A existuje nekonečně mnoho ekvivalentních automatů (automaty rozpoznávající stejný jazyk $L(A)$). Tyto automaty mohou například obsahovat spoustu nedosažitelných stavů, které jdou jednoduše algoritmicky odstranit. Algoritmus jednoduše prochází stavy z počátečního stavu pomocí přechodové funkce, přitom každý stav může navštívit vždy jen jednou. Zbylé stavy, do kterých se nedostane, jsou tzv. nedosažitelné a z automatu je můžeme odstranit. V této kapitole se ale zaměřím hlavně na algoritmus odstraňující stavy, které jsou dosažitelné, ale vlastně nadbytečné. Přesné označení nadbytečného stavu popíšu v následujícím příkladu 2 automatů



Obrázek 5: První Automat v příkladu pro redukci automatu



Obrázek 6: Druhý Automat v příkladu pro redukci automatu

Na obrázcích číslo 5 a 6 můžeme vidět dva automaty, které se na první pohled od sebe odlišují, ale nakonec poznáme, že rozpoznávají stejný jazyk a to přesně $L = \{w \in \{a, b\}^* | w$ má lichou délku$\}$. Hned si také všimneme na Obrázku 5 stavu 3, který můžeme nazvat nadbytečný a radši si vybereme řešení, které je na Obrázku 6. Pro přesnost můžeme nadefinovat jazyk L_q^{toAcc}, obsahující slova, které při čtení, jenž začíná ve stavu q, skončí v přijímajícím stavu. Poté můžeme

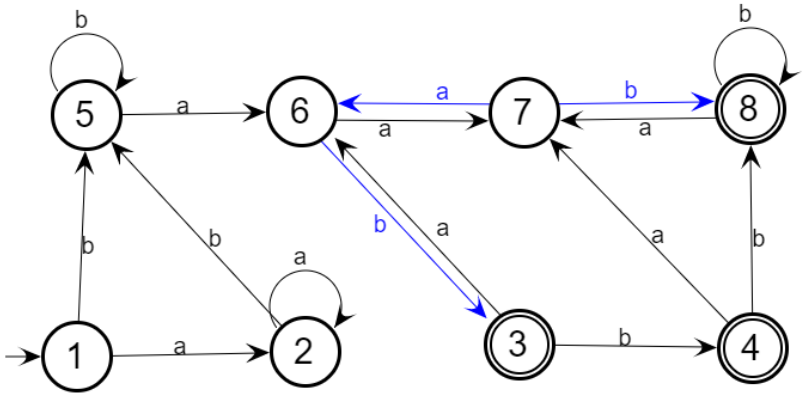
u tohoto příkladu zapsat $L_1^{toAcc} = L_3^{toAcc}$, což znamená, že stav 3 zde vlastně plní stejný úkol jako stav 1. V automatech těchto stavů, plnící stejný úkol, můžeme samozřejmě najít mnohem více. Z toho plyne, že ke každému automatu existuje ekvivalentní automat, který můžeme nazvat redukovaný.

Definice 5 [1]

Konečný automat je redukovaný, jestliže neobsahuje nedosažitelné stavy a pro každé dva různé stavy q, q' platí $L_q^{toAcc} \neq L_{q'}^{toAcc}$.

Z toho plyne, že potřebujeme umět u obecného automatu zjišťovat, zda pro dva stavy q, q' platí $L_q^{toAcc} \neq L_{q'}^{toAcc}$. S tím se dostávám k hlavnímu algoritmu, který toto dokáže pro všechny dvojice stavů. Algoritmus popíšu slovně přímo na příkladu.

Př. Mějme automatu popsaný následujícím grafem:



Obrázek 7: Automat pro příklad redukce automatů

Algoritmus začíná pomocí hned jasného pravidla, kdy nemůže platit $L_q^{toAcc} = L_{q'}^{toAcc}$, pokud jeden ze stavů je přijímající a druhý ne (jeden z jazyků obsahuje ε a druhý ne). Jinými slovy, takové dva stavy rozlišíme již slovem délky 0. Množina stavů v tomto případě {1, 2, 3, 4, 5, 6, 7, 8} se rozpadá na dvě disjunktní podmnožiny {1, 2, 5, 6, 7} (nepřijímající stavy) a {3, 4, 8} (přijímající stavy). První množinu pak označíme jako třída I a druhou jako třída II. Tyto třídy se na začátku odlišují jenom tím, že první třída obsahuje stavy, které nepřijímají slova délky 0, a naopak druhá třída tento jazyk přijme.

Dále musíme zjistit rozdíl při délce slova nejvýše 1. K tomu nám poslouží tabulka automatu, s tím, že místo konkrétních stavů píšeme do políček tabulky jen označení příslušné třídy, do které se dostaneme pomocí přechodové funkce. Stavy teď píšeme v pořadí tříd.

		a	b
I	1	I	I
	2	I	I
	5	I	I
	6	I	II
	7	I	II
II	3	I	II
	4	I	II
	8	I	II

Tabulka 7: Tabulka v prním kroku redukce automatu

Z tabulky je ihned vidět, že např. stavy 1 a 6 (které nelze rozlišit slovem délky 0) lze rozlišit slovem délky 1. A proto se L_1^{toAcc}, L_6^{toAcc} neshodují už na slovech délky 1. Z toho vyplývá, že jazyky L_q^{toAcc} a $L_{q'}^{toAcc}$ se shodují slovem délky nejvýše 1, právě tehdy když q a q' patří do stejné třídy rozkladu. Tímto rozkladem jsme dostali třídy $I = \{1, 2, 5\}$, $II = \{6, 7\}$ a $III = \{3, 4, 8\}$. Další rozklad dostaneme logicky z těchto tříd.

		a	b
I	1	I	I
	2	I	I
	5	II	I
II	6	II	III
	7	II	III
III	3	II	III
	4	II	III
	8	II	III

Tabulka 8: Tabulka v druhém kroku redukce automatu

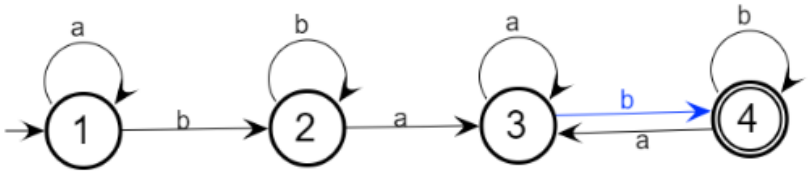
Zde vidíme rozdíl jenom ve třídě I u stavu 5 oproti 1. Znovu opakujeme postup rozdělení na třídy. V tomto případě se rozdělí na $I = \{1, 2\}$, $II = \{5\}$, $III = \{6, 7\}$, $IV = \{3, 4, 8\}$.

		a	b
I	1	I	II
	2	I	II
II	5	III	II
II	6	III	IV
	7	III	IV
IV	3	III	IV
	4	III	IV
	8	III	IV

Tabulka 9: Tabulka v druhém kroku redukce automatu

Na první pohled už vidíme, že se třídy dále nedělí. Dosáhli jsme bodu, kdy konstrukce dosáhla finálního rozkladu. Tedy, pokud se v tomto automatu jazyky L_q^{toAcc}, $L_q'^{toAcc}$ liší, pak

se liší již na slovech délky nejvýše dvě. Následně stačí z každé třídy nechat jediného zástupce a všechny přechody nasměřujeme na tohoto zástupce. Počáteční stav je samozřejmě ta třída, která obsahuje původní počáteční stav. Třída je pak přijímající jenom tehdy, pokud obsahuje jenom přijímající stavy z počátečního automatu (už u prvního rozkladu byly v třídách všechny stavy buď přijímající, nebo nepřijímající a poté to platí u každého následujícího rozkladu).



Obrázek 8: Výsledný graf automat z příkladu pro redukci automatu

2.8 Automaty pro průnik a sjednocení jazyků

Sjednocení a průnik jsou první z uzavřených operacích nad třídou regulárních jazyků, kterými se zde budu zabývat. Uzavřenost operací znamená, že pokud je jazyk regulární, bude i jazyk, který dostaneme z těchto operací, také regulární.

Definice 6 *Průnik:*

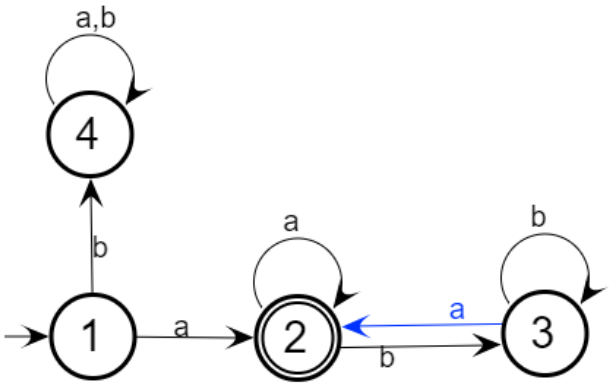
$L_1 \cap L_2 = \{w | w \in L_1 \ a \ w \in L_2\}$ - *výsledný jazyk se skládá ze slov, které se nachází jak v jazyku* L_1*, tak v jazyku* L_2

Definice 7 *Sjednocení:*

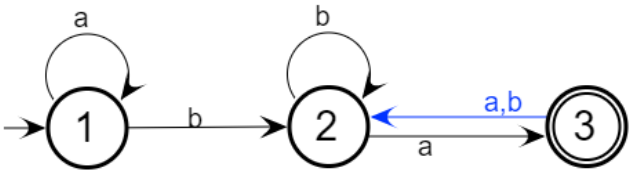
$L_1 \cup L_2 = \{w | w \in L_1 \ nebo \ w \in L_2\}$ - *výsledný jazyk se skládá ze všech slov jak jazyka* L_1*, tak* L_2

Algoritmus pro sjednocení a průnik dvou deterministických automatů pak vypadá podobně jako algoritmus pro převod nedeterministického na deterministický automat, přitom výsledné stavy jsou uspořádané dvojice stavů původních automatů (první je stav z automatu prvního a druhý z druhého). Algoritmus začíná dvojicí počátečních stavů obou automatů a prochází pomocí přechodové funkce všechny možné kombinace dvou stavů, do kterých se dostane. Rozdíl pak v průniku oproti sjednocení spočívá v přijímajících stavech, kdy pokud se jedná o sjednocení, tak výsledný stav je přijímající, pokud aspoň jeden ze stavů (stav z automatu prvního, nebo automatu druhého) je přijímající, pokud se jedná o průnik, musí být oba dva stavy přijímající.

Př. Mějme automat rozpoznávající jazyk $L1 = \{w \in \{a, b\}^* | w$ začíná i končí znakem $a\}$ a automat s jazykem $L2 = \{w \in \{a, b\}^* | w$ má sufix $ba\}$. Nad těmito automaty uděláme operaci sjednocení. Graf prvního automatu je pak na Obrázku 9 a druhý na Obrázku 10.



Obrázek 9: První automat pro příklad sjednocení automatů



Obrázek 10: Druhý automat pro příklad sjednocení automatů

Na začátku je algoritmus v počátečním stavu prvního automatu 1 a druhého 1, ze kterých se vytvoří počáteční stav pro výsledný automat jako uspořádaná dvojice (1, 1). Stav je pak v operaci sjednocení přijímající, právě tehdy, pokud aspoň jeden ze stavů je přijímající (v našem případě přijímající není). Stejně jako v příkladu pro převod NKA na DKA si zde pomůžeme převodovou tabulkou, do které počáteční stav zapíšeme a rovnou zapíšeme výsledky přechodové funkce vedoucí z toho stavu (v tomto příkladu znakem a do dvojice stavů (2, 1) a znakem b do dvojice (4,2)).

	a	b
→ (1, 1)	(2, 1)	(4, 2)

Tabulka 10: Tabulka v prním kroku sjednocení

V dalším kroku se podíváme na stavy do kterých přechodové funkce směřují, a pokud je v tabulce už nemáme (v našem případě se ani jeden stav nerovná počátečnímu stavu), připíšeme je na nový řádek, jako nové stavy ve výsledném automatu. Poté k nim jako v minulém kroku připíšeme další přechodové funkce. Musíme si také všimnout, že stav 2 v prvním automatu je přijímající, proto bude stav (2, 1) také přijímající.

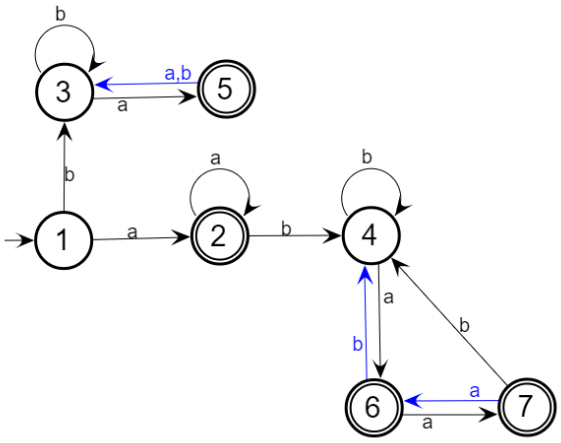
	a	b
→ (1, 1)	(2, 1)	(4, 2)
← (2, 1)	(2, 1)	(3, 2)
(4, 2)	(4, 3)	(4, 2)

Tabulka 11: Tabulka v druhém kroku sjednocení

Dále postupujeme stejně jako v předešlém kroku. V tomto případě musíme vytvořit nové 2 stavy (3, 2) a (4, 3). Takto postupujeme do té doby, než výsledný automat bude deterministický. Pro přehlednost do výsledné tabulky připíšu nové pojmenování stavů, tak aby se shodovaly i s výsledným grafem automatu.

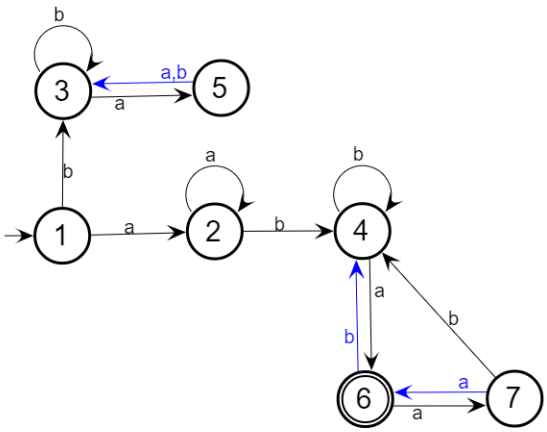
	a	b
→ 1 : (1, 1)	(2, 1)	(4, 2)
← 2 : (2, 1)	(2, 1)	(3, 2)
3 : (4, 2)	(4, 3)	(4, 2)
4 : (3, 2)	(2, 3)	(3, 2)
← 5 : (4, 3)	(4, 2)	(4, 2)
← 6 : (2, 3)	(2, 2)	(3, 2)
← 7 : (2, 2)	(2, 3)	(3, 2)

Tabulka 12: Tabulka v posledním kroku sjednocení



Obrázek 11: Výsledný graf automatu v příkladu sjednocení

Pro porovnání zde přidám i výsledek pro průnik těchto dvou automatů. Zde je změna jenom v ohodnocení, zda má být výsledný stav přijímající nebo ne.



Obrázek 12: Výsledný graf automat z příkladu pro průnik

Na tomto automatu jde vidět, že se dá dále zredukovat (např. stav 5 má stejnou úlohu jako stav 3). Také se hned všimneme rozdílu mezi průnikem a sjednocením, kde u průniku máme jenom jeden stav přijímající.

Zde ještě zmíním i další jednodušší postup sjednocení, který ale platí pouze pro nedeterministické automaty. U nich můžeme využít toho, že NKA může být v jednu chvíli ve více stavech. Tato metoda spočívá ve vytvoření nového počátečního stavu, ze kterého vedeme epsilon přechody do počátečních stavů automatů, nad kterými chceme průnik provést. Poté je zajištěno, že pokud slovo patřilo aspoň do jednoho z automatů, tento nový automat ho přijme, protože předešlý automat je součástí nového automatu a slovo předešlým automatem pak projde celé.

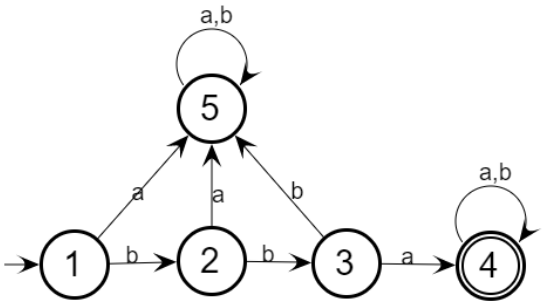
2.9 Zrcadlový obraz

Další operací, kterou se zde budu zabývat, je zrcadlový obraz jazyka. Nejdřív začnu definicí zrcadlového obrazu a poté rovnou přejdu k příkladu pro pochopení algoritmu, který z automatu vytvoří jiný automat rozpoznávající zrcadlový obraz předešlého automatu.

Definice 8 [1]

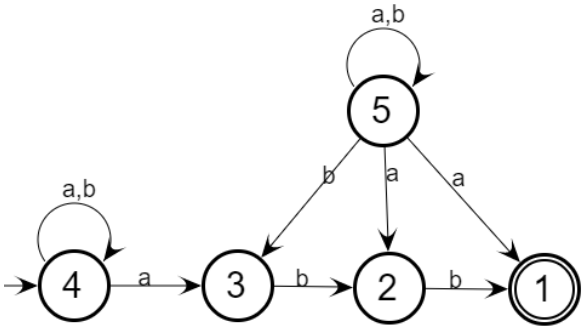
Zrcadlový obraz slova $u = a_1a_2.....a_n$ *je* $u^R = a_na_{n-1}....a_1$*, zrcadlový obraz jazyka* L *pak je* $L^R = \{u|\exists v \in L : u = v^R\}$*, stručněji psáno* $L_1^R = \{u^R|u \in L\}$.

Př. Mějme automat popsaný grafem



Obrázek 13: Graf automatu pro příklad zrcadlového obrazu

Na tomto příkladě ukážu jednoduchost algoritmu, který využívá vlastnosti nedeterministických automatů (i když je zřejmé, že automat je na první pohled deterministický, je zároveň i nedeterministický, protože všechny DKA jsou zároveň i NKA). Algoritmus nejdřív změní orientaci přechodů v grafu. Poté jednoduše změní počáteční stavy na přijímající (v tomto příkladu jenom stav 1) a přijímající naopak na počáteční (zde jenom stav 4). Graf automatu pak vypadá jak na dalším obrázku.



Obrázek 14: Graf výsledného automatu příkladu zrcadlového obrazu

Zde vidíme, že výsledný automat přijímá slova končící na *abb*, což je zrcadlový obraz minulého automatu, který rozpoznával slova jazyka začínající na *bba*.

2.10 Doplněk

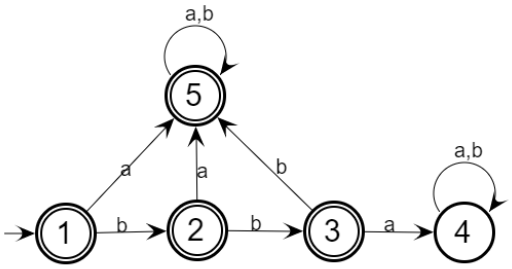
Poslední operací nad jazyky, která je součástí této diplomové práce je doplněk. Algoritmus pro tuto operaci je nejjednodušší, a proto začnu rovnou definicí a poté algoritmus předvedu na příkladu.

Definice 9 *Doplněk nad regulárním jazykem*

Pro jazyk L je jeho doplněk $\bar{L}$, kde $\bar{L} = \Sigma^* - L$. tzn. slova patřící do jazyka $\bar{L}$ jsou ty, které nepatří do jazyka L.

Př. Pro příklad doplňku si vezmu příklad z minulé operace zrcadlový obraz tzn. automat vyobrazený grafem na Obrázku 13.

Algoritmus při doplňku si jednoduše u deterministického automatu přehodí stavy přijímající na nepřijímající a naopak (v tomto příkladu máme jenom jeden stav přijímající, takže výsledek bude, že ostatní stavy budou přijímající a onen stav 5 přijímající nebude). Výsledek je níže vyobrazen znovu grafem.



Obrázek 15: Graf výsledného automatu příkladu doplněk

3 Analýza požadavků

Tato kapitola se věnuje analýze požadavků nově vzniklého softwarového řešení. Jelikož je výstupem této diplomové práce serverová aplikace kladoucí si za cíl pomoct studentům předmětu Teoretická informatika při jejich studijním úsilí, vycházel jsem v této analýze, kromě požadavků vedoucího práce, také ze svých zkušeností s tímto předmětem.

3.1 Funkční požadavky

Základní funkční požadavky byly vymezeny v zadání práce, další, specifičtější, byly postupně přidávány při konzultacích s vedoucím práce.

3.1.1 Proč nový systém

Jak již bylo zmíněno v úvodu této kapitoly, vzniká tento systém z důvodu usnadnění pochopení látky předmětu Teoretická informatika, konkrétně části věnované konečným automatům. Tento systém umožní uživateli vizuálně pochopit postup algoritmů pracující s konečnými automaty.

3.1.2 K čemu má nový systém sloužit

Systém má sloužit k podpoře výuky teoretické informatiky. Hlavním cílem je ukázat na příkladu, který si uživatel přímo do systému sám zadá, postup algoritmů pro práci s konečnými automaty. Dále bude možné si příklady konečných automatů ukládat do souboru a vkládat je zpátky jako vstup pro další výpočet. Součástí bude i možnost do systému zadávat příklady, které budou uloženy na server a každý uživatel si je tak poté bude moct vyzkoušet.

3.1.3 Kdo bude se systémem pracovat?

Systém je určen hlavně pro studenty předmětu teoretická informatika a dalších předmětů, kde se vyučují konečné automaty. K užívání systému nejsou zapotřebí speciální přístupová práva, takže se systém budou moct využívat i nejrůznější cizí uživatelé. Systém by proto měl mít jednoduché a intuitivní ovládání. Noví uživatelé se v systému zorientují pomocí nápovědy, která bude ke stažení jako soubor pdf.

3.1.4 Jaké budou vstupy do systému?

Vstupy aplikace jsou následující:

- **Zadání automatu pomocí grafu**: uživatel bude moct zadat automat pomocí grafu, který bude vykreslován na HTML5 canvas, kde si také bude moct uspořádat jednotlivé stavy jednoduchým drag and drop (jednoduchá manipulace pomocí myši).

- **Zadání automatu přechodovou tabulkou**: další zadání automatu je pomocí tzv. přechodové tabulky, do které se dostane kliknutím na tlačítko, umožňující přepnutí z režimu zadávaní pomocí grafu. Toto zadávání se pak podobá jako zápis automatu právě pomocí přechodové tabulky.
- **Vložení automatu z uloženého souboru**: pro jednoduchost další simulace, některého algoritmu, je pak načtení z uloženého souboru typu XML, který zde v další kapitole podrobněji popíšu.
- **Načtení automatu z příkladu uloženého na serveru**: poslední zadání automatu je pro účely výuky načtení příkladu, který bude uložen přímo na serveru.

3.1.5 Jaké budou výstupy ze systému?

Algoritmy mají svoje specifické grafické (např. barevné označení stavů, se kterými v daném kroku algoritmy pracují) a textové výstupy (např. přímo popsaný krok, který algoritmus udělal) pro pochopení daného postupu, kde konečným výstupem bude, kromě simulace běhu na daném slovu, nový automat. Každý automat, buď zadaný uživatelem, nebo vypočítaný některým z algoritmů, se bude poté moct uložit jako XML soubor pro další použití.

3.1.6 Jaké bude mít systém funkce?

Systém bude řešit následující algoritmy: simulace běhu automatu na zadaném slově, převod nedeterministického automatu na deterministický, redukce automatu, průnik a sjednocení dvou automatů, zrcadlový obraz, doplněk.

3.2 Nefunkční požadavky

Nefunkční požadavky kladou důraz na design aplikace, odezvu, bezpečnost, použitý programovací jazyk, na použité standardy, atd.

3.2.1 Design aplikace

Je třeba, aby byla aplikace jednoduchá a intuitivní na používání. Design aplikace je pak čerpán z využívání nástrojů bootstrap. Hlavní menu, které se nachází v horní části aplikace, pak spojuje jednotlivé algoritmy.

3.2.2 Použité technologie

Server bude vyvíjen v prostředí Microsoft Visual Studio v programovacím jazyku C#. Jako webový aplikační Framework použiji ASP.NET MVC ve verzi 5 a poté pro komunikaci se serverem budu využívat AJAX a na straně serveru pak přesněji Framework ASP.NET Web API 2. K reálnému nasazení je pak třeba server s podporou technologie ASP.NET (nejlépe IIS).

3.2.3 Odezva aplikace

Rychlost odezvy je důležitým parametrem. Zde není potřeba, aby byl algoritmus výpočtu co nejrychlejší, ale je nepřípustné, aby uživatel dlouho čekal na výsledek výpočtu. Proto je důležité zvolit vhodné metody pro výpočet.

3.2.4 Použité standardy

Jedná se o webovou aplikaci a stránky tedy musí být přístupné uživatelům. Je důležité se postarat, aby uživatelé mohli na stránky přistupovat z různých prohlížečů a proto bude aplikace testována na nejpoužívanějších prohlížečích (Google Chrome, Mozilla Firefox, MS Edge).

4 Návrh

Následuje návrh struktury serveru a grafického rozhraní.

4.1 Architektura

Celý systém je programován pomocí technologie ASP.NET, která je součástí .Net Frameworku, na straně serveru a javascriptu na straně prohlížeče. Přesněji pak použiji z ASP.NET webový aplikační Framework MVC založen na softwarové architektuře Model-View-Controller. Pro komunikaci mezi serverem a prohlížečem při výpočtu dalších kroků v algoritmu využiji AJAX, kvůli změně obsahu bez nutnosti znovunačtení celého obsahu stránky. Na straně webového prohlížeče pak přesněji pro AJAX budu využívat javascriptové knihovny jQuery a na straně serveru pro komunikaci ASP.NET Web API 2, kvůli snadné implementaci a podobnosti s ASP.NET MVC. Pro vykreslování grafu automatu na vstup i výstup a následné manipulace s ním, zde budu používat HTML5 canvas.

4.2 Grafické uživatelské rozhraní

Následuje popis grafického rozhraní aplikace. Grafické rozhraní bude využívat soubor nástrojů bootstrap.

4.2.1 Hlavní menu

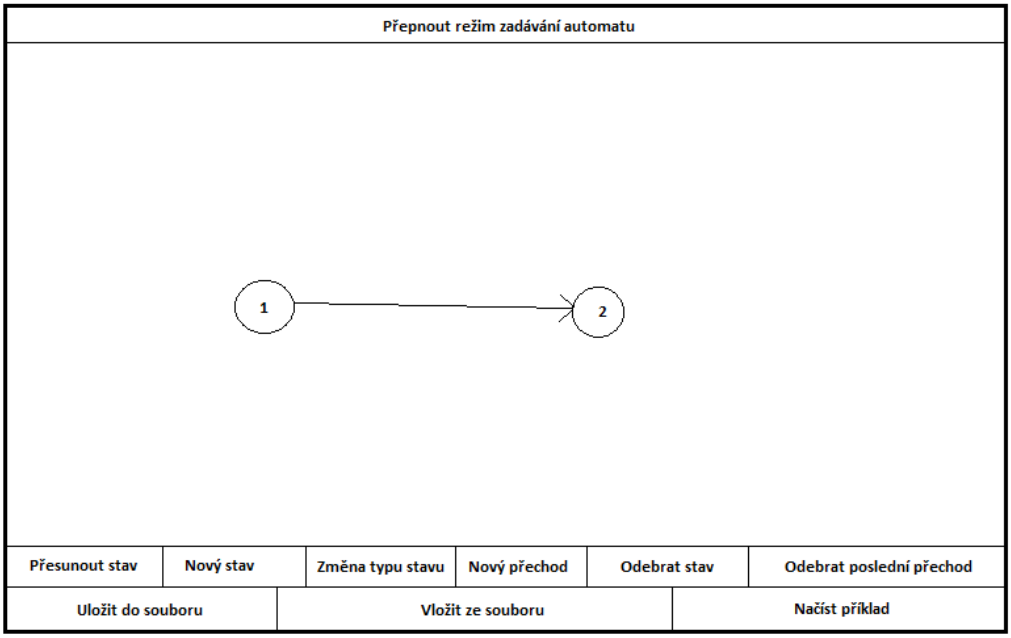
Hlavní menu webové aplikace s hlavním účelem co nejlehčího přechodu mezi samotnými algoritmy, bude orientované na horní části aplikace. Zde bude také odkaz na stránku s nápovědou a odkaz na domovskou stránku aplikace.

4.2.2 Popis vstupu a výstupu automatu

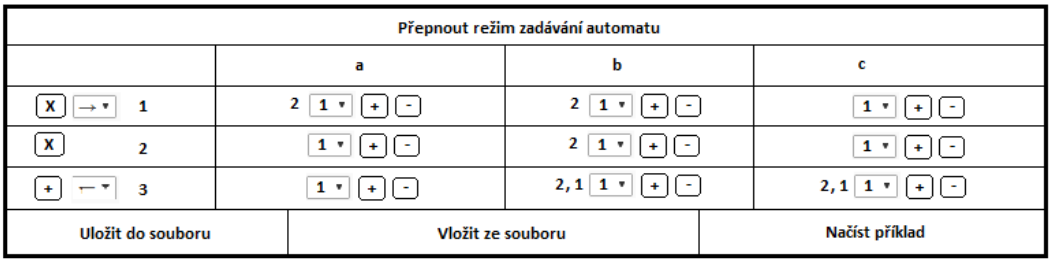
Skládá se z jedné komponenty, ve které bude možno zadvání automatu uživatelem dvěma způsoby. Tyto způsoby jsou navzájem propojené (tzn. pokud zadání proběhne jedním vstupem, objeví se i na vstupu druhém). Mezi vstupy bude možné snadno přepínat.

- **Zadání pomocí grafu**: Tato varianta bude využívat pro zobrazení HTML5 canvas, na kterém se budou zobrazovat graficky stavy s přechody mezi nimi. Stavy se budou moct pomoci myši přemísťovat (funkce drag/drop). Pro vytvoření a mazání stavu budou využity tlačítka pod plátnem.
- **Zadání pomocí přechodové tabulky**: Jednoduchá manipulace pomocí vstupních HTML komponent vytvářejících tabulku, kde u každého stavu bude na výběr typ stavu (přijímající, nepřijímající, počáteční) a v políčkách bude možnost přidání nového přechodu do jiných stavů. Vložení nového stavu bude řešeno posledním řádkem tabulky.

Na spodní části celé této komponenty budou tlačítka na uložení automatu do XML souboru, dále pro načtení automatů z XML a načtení automatu z příkladu uloženého na serveru (tyto tlačítka budou v obou režimech zadávání).



Obrázek 16: Rozložení komponenty automatu (režim zadávání grafem)



Obrázek 17: Rozložení komponenty automatu (režim zadávání grafem)

4.2.3 Popis stránek s algoritmy

Dále následují návrhy stránek pro výuku samotných algoritmů. Všechny stránky budou mít na začátku stručný popis, o jaký algoritmus se jedná a pod ním si budou moct vybrat ze dvou abeced, kdy se po přepnutí celá stránka znovu načte se vstupy pro nový automat obsahující tuto abecedu.

Zde je jednoduchý slovní popis všech stránek:

- **Simulování běhu na zadaném slově**: v menu jednoduše „Simulování běhu“. Skládá se z textového pole pro zadání slova na simulaci, tlačítka pro začátek simulace a hlavní komponenty pro zadání automatu (popíšu níže). Po kliknutí na tlačítko **Start**, tlačítko zmizí a objeví se tabulka s průběhem simulace, další tlačítka pro manipulaci průběhu simulace (**Další**, **Zpět** a **Zrušit**). Dále se při simulaci bude barevně označovat průběh přímo na grafu automatu.

- **Redukce automatu**: První je komponenta pro zadání vstupního automatu, poté následuje tlačítko pro začátek redukce (**Start**). Po stisknutí tohoto tlačítka se objeví tabulka rozkladu tříd a šipka na následující krok. Na konci je výsledný (zredukovaný) automat vykreslený jako další komponenta pro vyobrazení automatu grafem i tabulkou zároveň.

- **Převod nedeterministického konečného automatu na deterministický**: v menu pod NKA na DKA. Komponenta pro zadání nedeterministického vstupního automatu, dále následuje tlačítko **Start**. Po stisku **Start** se objeví nový automat, který bude vizuálně ukazovat vytváření deterministického automatu ze vstupního. Součástí je pak menší tabulka ukazující převod označení nových stavů na stavy předešlého automatu a tlačítka pro manipulaci mezi jednotlivými kroky (**Další** a **Zpátky**). Pod výsledným automatem bude ještě textově popsáno, co v jakém kroku algoritmus provedl.

- **Průnik dvou automatů**: Tato stránka začíná vstupem pro první a pod ním pro druhý automat, následuje **Start**. Na začátku se objeví další komponenta pro automat, vykreslující výsledný automat, s tlačítky pro řízení přechodu mezi jednotlivými kroky a tabulka s převodem (mezi stavy výsledného automatu a stavy dvou vstupujících automatů), jako u převodu NKA na DKA.

- **Sjednocení dvou automatů**: Zde je vstup i výstup stejný jako u průniku dvou automatů

- **Zrcadlový obraz automatu**: První komponenta je vstupní automat, dále tlačítko pro začátek. V prvním kroku se objeví automat jenž ho popisuje, dále tlačítko pro další krok. Druhý a zároveň poslední krok je opět zakončen automatem.

- **Doplněk**: Podobné předchozímu zrcadlovému obrazu, kromě posledního kroku, protože zde stačí jenom krok jeden.

4.2.4 Uživatelská příručka

Vzhledem k tomu, že se jedná o server pro podporu výuky konečných automatů, je nutné, aby studenti měli k dispozici uživatelskou příručku, pomocí které se seznámí hlavně s komponentou pro vytváření automatu jak už pomocí grafu, tak pomocí přechodové tabulky.

5 Popis implementace

Implementace probíhala průběžně v iteracích na základě pravidelných konzultací s vedoucím diplomové práce. Nejdřív začnu stručným vysvětlením použitých technologií, a pak se zaměřím hlavně na implementaci projektu.

Celou aplikaci zde rozdělím na část klientskou (část s logikou na straně klientského webového prohlížeče), skládající se hlavně z komponenty pro vstup i výstup konečného automatu, a na část serverovou, kde probíhá výpočet po krocích nad daným algoritmem.

Algoritmy začnu popisovat nejdřív modelem, který slouží k přenášení dat mezi prohlížečem a serverem a pak slovy popíšu implementaci algoritmu s tím, jak se vykresluje uživateli v prohlížeči, krok po kroku. Všechny modely budu vykreslovat pomocí UML diagramu tříd. Při vykreslení diagramu modelu pro algoritmy nebudu znovu do diagramu zapisovat model automatu, ale rovnou ho budu používat jako typ u properties (dále jen česky vlastnosti). Nakonec pro složitější algoritmy přidám diagram aktivit pro stručný popis procesů na serveru.

5.1 Použité technologie

5.1.1 Vývojové prostředí

Jako vývojové prostředí jsem zvolil Microsoft Visual Studio 2015, kvůli implementaci pomocí jazyka C# na platformě .Net (přesněji pak ASP.NET MVC a ASP.NET WEB API 2) a kvůli zkušenostem s tímto vývojovým prostředím. Výhodou toho prostředí je široká škála podporovaných jazyků (např. Visual C++, C#, VB.NET, F#, atd.) a vývoje pro různé druhy frameworků (např. zmíněný ASP.NET MVC, ASP.NET WEB API 2 a také desktopové aplikace mezi které patří například Windows Presentation Foundation) a zařízení (např. Windows Mobile, Windows CE). Další výhody pro vývoj webových aplikací v tomto prostředí je jednoduchost debugování a intellisense (podporující jak programovací jazyk C#, tak i XML, CSS a Javascript pro vývoj webových stránek).

5.1.2 ASP.NET MVC

Je webový aplikační framework implementující vzor Model-View-Controller (MVC) postavený na .Net frameworku.

Stručné vysvětlení vzoru MVC:

- **Model**: objekt, který má za úkol přenášet data mezi Controllerem a View (v jiných projektech se často používá rovnou pro mapování dat z databáze do těchto objektů, který se poté rovnou vypisuje ve View). Třídy těchto objektů jsou v projektu ve složce Models.
- **View**: je komponenta představující uživatelské rozhraní (typicky pak toto UI je vytvořeno z dat v modelu). Soubory view jsou pak ve složce Views.

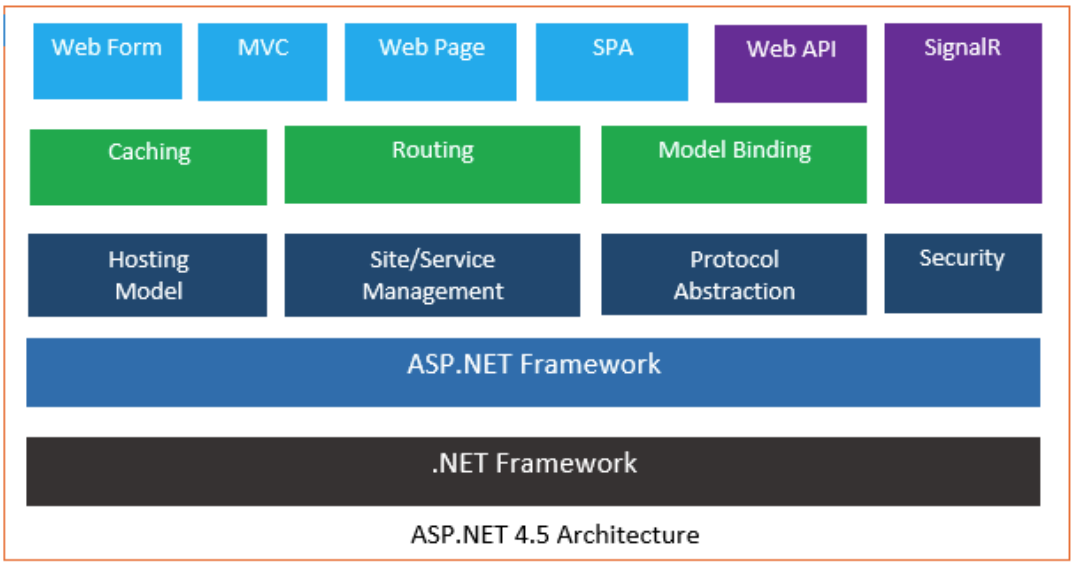
- **Controller**: hlavní komponenta, která se stará o logiku aplikace na straně serveru. Zachycuje interakci uživatele (např. uživatel stiskne tlačítko pro potvrzení HTML formuláře a data poté Contoller odchytí a uloží do databáze). Pracuje s modelem, který pak předá View (zde vždy nemusí volat View, ale data rovnou poslat uživateli na výstup třeba ve formě JSON).

5.1.3 ASP.NET WEB API 2

Je Framework pro vytváření RESTful aplikací založený na .Net frameworku. REST je architerktura rozhraní, navržená pro tvorbu webových služeb (využívá http protokol). Na rozdíl od SOAPu, který volá procedury (přímo v těle zprávy je jméno procedury, kterou volá), REST služby jsou orientované na zdroje (stručněji je pak jméno procedury rovnou v URL a v těle zprávy se přenášejí jenom data).

Samotné ASP.NET WEB API je pak podobné implementaci ASP.NET MVC s rozdílem, že neobsahuje View a místo něho předává data obsažené v Modelu ve formátu JSON nebo XML.

Pokud je projekt založen jako ASP.NET Web Application můžou poté tyto technologie být součástí jednoho projektu.



Obrázek 18: Architektura ASP.NET [3]

Pro všechny tyto technologie od společnosti Microsoft existují na stránkách MSDN [4] dokumentace, které v této práci nejčastěji využívám.

5.2 Vstup a výstup automatu

V této části popíšu jak vytváření automatu pomocí grafu a tabulky, tak i vytváření souboru XML, pro uložení a pozdějšího vložení stejného automatu na vstup.

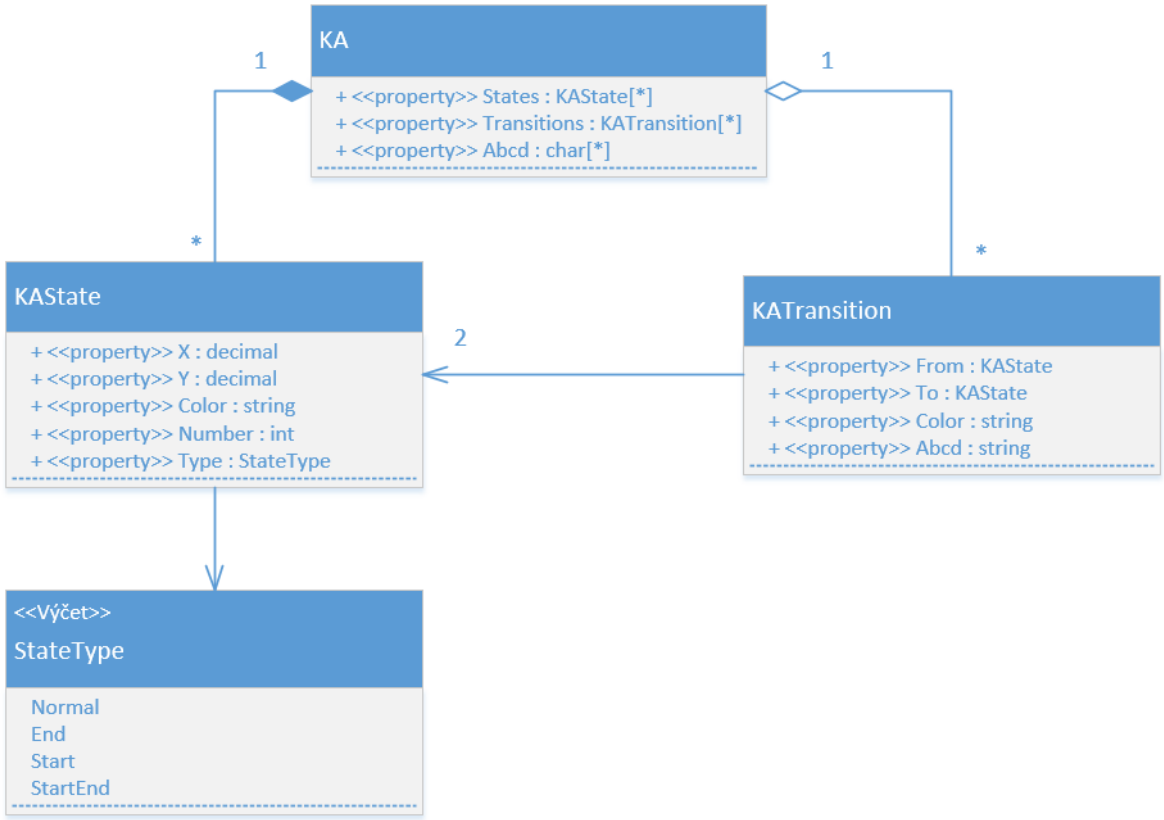
Některý z algoritmů vyžaduje, aby automat byl deterministický. Proto se dá komponenta pro vstup a výstup automatu nastavit jen pro deterministický automat. Toto se projeví na grafu tím, že například pokud se bude uživatel snažit přidat nový přechod znakem, který už stav jeden má, přechod se nevloží, nebo u vložení automatu pomocí přechodové tabulky, se tlačítka, pro vložení nového přechodu, přepnou do stavu vypnuté (disabled).

Dále se vstupy musí dát vypnout (tzn. Schovat tlačítka nebo jiné HTML elementy pro vstup nových stavů a přechodů), protože při chodu algoritmů by se nemělo dále manipulovat s automatem jak vstupním tak automatem výstupním.

Ze začátku se zaměřím na pochopení, jak vypadá model popisující automat a poté přejdu k vykreslování tohoto modelu jako grafu na canvas a do tabulky. Následně popíšu mapování modelu do XML souboru, tak aby se dal automat uložit a znovu načíst z tohoto souboru.

5.2.1 Model automatu

Pro manipulaci automatu jak na straně serveru, tak na straně klientského prohlížeče v Javascriptu slouží model, který je popsán jako třída na straně serveru. Klient tento model dostane rovnou jako javascriptový objekt (JSON) a s ním pak může jednoduše pracovat.



Obrázek 19: Model KA

Jak už názvy vlostností u třídy `KA` napovídají, `States` jsou stavy automatu, `Transitions` jsou přechody (přechodové funkce) a poslední je abeceda celého automatu `Abcd`. Zde jsem volil typ char, aby se výsledná abeceda skládala z přesných (vždy jen jeden symbol na jeden znak) znaků. Dále popíšu třídy `KAState` a `KATransition`.

U `KAState` vlastnosti `X` a `Y` slouží k uchování pozice stavu na HTML5 canvasu na straně klientského prohlížeče a na serveru se s nimi pracuje jenom při ukládání a načítání stavu ze souboru. Jinak na klientské části, pokud tyto vlastnosti při vykreslování do canvasu chybí, pozice stavu se volí náhodně a uživatel si pak stav sám přesune na vhodné místo. `Color` u stavu i přechodu znamená, jak už pojmenování napovídá, barvu se kterou se vykresluje uživateli (používá se hlavně pro zvýraznění daného kroku v samotných algoritmech). `Number` pak v `KAState` je hlavní pojmenování stavu. Zde jsem volil integer kvůli nejjednodušší iteraci při vkládání nového stavu a celkového seřazení, i když poté se stavy budou muset pojmenovávat jedině pomocí čísla. Další je `Type` v `KAState`, který udává podle výčtového typu StateType, typ stavu.

- **Normal**: stav, který není ani přijímající a ani počáteční
- **End**: stav přijímající
- **Start**: stav počáteční
- **StartEnd**: stav jak přijímající, tak počáteční

V `KATransition` `From` a `To` s typem `KAState` je poté jasné, že se jedná o stavy od jakého a do jakého stavu se přechod vede. Zde se musí dávat pozor, že pokud se budou data posílat z prohlížeče na server a zpátky, nebude objekt ve vlastnostech `From` a `To` stejný jako objekt v poli `States` a musí se porovnávat přímo vlastnost `Name` v daných objektech třídy `KAState`. Poslední vlastnosti jsou `KATransition` a `Abcd` popisující znaky přechodu. Přechod se může sestávat z více znaků a proto jsem zde zvolil string, kde se znaky oddělují čárkou.

5.2.2 Vykreslovaní grafu automatu na HTML5 canvas

Pro vykreslování grafu, tak aby se s tímto grafem dalo jenoduduše manipulovat pomocí myši (funkce drag/drop nad vykreslenými objekty, v tomto případě vykreslené stavy) a dalších prvků, jsem zvolil HTML5 element canvas. Tento prvek slouží k dynamickému vykreslování bitmap pomocí Javascriptu za použití mnoha kreslících funkcí podobných jiným běžným 2D API. Canvas sestává z regionu, na který lze kreslit, definovaného v HTML kódu šířkou a výškou. Dále se dá využít i pro vykreslování ve 3D s WebGL založeným na OpenGL.

V projektu poté, pro vykreslování grafu do canvasu a manipulaci s celým automatem slouží javascriptový soubor canvasKA.js

Pro vykreslování objektů na daný canvas se používá jeho context, který se bere z elementu pomocí funkce *getContext(contextType)*, kde contextType je typ contextu (např. 2d pro jednoduché vykreslování 2d grafiky, nebo webgl pro vykreslování složitější 3d grafiky). V této diplomové

práci bude stačit 2d context. V tomto contextu, pokud dojde ke změně nějakého objektu (např. změna pozice stavu), musí se celá grafika v canvasu přepsat na novo. Pro tento účel slouží funkce drawShapes, která přepíše celý canvas hodnotami automatu předaného v parametru.

Stav

Pro vykreslení stavu, který není přijímající a ani počáteční, bude stačit vykreslit kolečko pomocí funkce na contextu `arc(x, y, r, sAngle, eAngle, counterclockwise)`, kde `x` je poloha centru kruhu v ose x, `y` je je poloha v ose y, `r` je poloměr, `sAngle` je začátek úhlu v radiánech a `eAngle` je konec úhlu, `counterclockwise` je pak volitelný parametr, který udává, jakým směrem se bude kolečko vykreslovat (výchozí hodnota je false, kdy se vykreslí ve směru hodinových ručiček).

```
context.fillStyle = stateToDraw.Color;
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad, 0, 2 * Math.PI);
context.closePath();
context.fill();
context.fillStyle = "rgb(255,255,255)";
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad - 3, 0, 2 * Math.PI);
context.closePath();
context.fill();
```

Výpis 1: Vykreslení kolečka jako stavu

V tomto kódu můžeme vidět, že se nejdřív vykresluje vnější kruh nadefinovaný barvou stavu a poté se vykreslí další kolečko bílou barvou. Do tohoto kolečka se ještě musí vykreslit číslo daného stavu.

```
context.fillStyle = stateToDraw.Color;
context.font = "30px Arial";
context.beginPath();
context.fillText(stateToDraw.Number, (stateToDraw.Number / 10 >= 1) ?
    stateToDraw.X - 17 : stateToDraw.X - 10, stateToDraw.Y + 10);
context.closePath();
context.fill();
```

Výpis 2: Vykreslení textu do canvasu

Zde dochází k vyplnění barvy na stejnou hodnotu jako má mít stav, protože minulá barva byla nastavena na bílou. Dále se vyplní velikost a styl písma a následně pomocí funkce `fillText(text, x, y, maxWidth)`, kde text je přímo `text`, který se má vypsat, `x` je poloha v ose x, `y` poloha

v ose y a `maxWidth` je volitelný parametr, který udává maximální délku textu v pixelech. Pokud je číslo už dvouciferné musí se samozřejmě posunout v ose x.

Stav, pokud je přijímající musí být označen ještě jedním kruhem ve vnitřní části stavu, proto je kód trochu delší, ale podobný kódu vypisující normální stav.

```
context.fillStyle = stateToDraw.Color;
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad, 0, 2 * Math.PI,
    false);
context.closePath();
context.fill();
context.fillStyle = "rgb(255,255,255)";
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad - 3, 0, 2 * Math.PI,
    false);
context.closePath();
context.fill();
context.fillStyle = stateToDraw.Color;
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad - 5, 0, 2 * Math.PI,
    false);
context.closePath();
context.fill();
context.fillStyle = "rgb(255,255,255)";
context.beginPath();
context.arc(stateToDraw.X, stateToDraw.Y, stateToDraw.Rad - 7, 0, 2 * Math.PI,
    false);
context.closePath();
context.fill();
```

Výpis 3: Vykreslení přijímajícího stavu

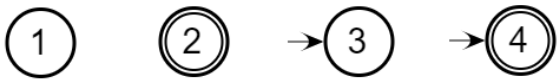
Poté se musí ještě napsat číslo stavu, jako u stavu normálního.

Pokud je stav počáteční, označuje se ještě šipkou směřující do něho. Při vykreslování šipek jsem si pomohl už naprogramovanou knihovnou s názvem canvasutilities.js, která je dostupná na adrese DBP Consulting [5], a kde je také dokumentace pro tuto knihovnu. Samotná knihovna dokáže lehce vytvořit šipky nad 2d contextem a to pomocí funkce `drawArrow(x1, y1, x2, y2, style, which, angle, length)`, kde `x1` a `y1` jsou souřadnice začínající přímku, `x2` a `y2` jsou souřadnice kde přímka končí, `style` typ hlavičky šipky (v tomto projektu volím nadefinovaný typ 4). Pomocí `angle` se pak nastavuje ostrost hlavičky šipky a poslední paramter `length` udávající velikost hlavičky šipky v pixelech.

```
context.fillStyle = defaultColor;
context.strokeStyle = defaultColor;
var fromX = stateToDraw.X - (stateToDraw.Rad * 2);
var fromY = stateToDraw.Y;
var toX = stateToDraw.X - stateToDraw.Rad;
var toY = stateToDraw.Y;
drawArrow(context, fromX, fromY, toX, toY, 4, 1, Math.PI / 8, 20);
```

Výpis 4: Šipka pro označení počátečního stavu

Proměnná *fillStyle* je zde v roli barvy hlavičky a *strokeStyle* je barva přímky.
Vykreslené stavy jsou pro ukázku zobrazeny na následujícím obrázku níže (Obrázek 19)



Obrázek 20: Ukázka stavů na výstupu canvasu

Přechod

Přechody podle modelu, který jsem popsal v podkapitole výše, se vykreslují z jednoho stavu do druhého. Tento postup potřebuje, aby se objekty v poli `States` shodovali s objekty u těchto přechodů (reference `From` a `To` u přechodu). Proto pokud přijde automat jako odpověď z webové služby (při výpočtu kroku v některém z algoritmů), musí se nejdřív propojit stavy z těmito přechody. Na to slouží funkce `bindTransitionsToState` s dvěma parametry `states` a `transitions`, což jsou jednoduše pole objektů KAState a KATransition. Funkce jednoduše nahradí objekt v referenci `From` a `To` v přechodu za objekt v poli `States`. Pro vykreslení celého přechodu se pak používá funkce *drawTransition*.

Dále se pro výkres šipky znovu používá stejná knihovna canvasutilities.js, jako u počátečního stavu. Zde nestačí dát jako konečnou souřadnici souřadnice stavu, do kterého přechod směruje, ale musí se vypočítat souřadnice bodu na hraně vnějšího kolečka stavu. Protože je přechod přímka se šipkou na konci, musí se vypočítat souřadnice konce přímky. K tomu použiji výpočet funkce přímky z dvou známých souřadnic (souřadnice automatu od a souřadnice automatu do) a poté pomocí Pythagorovy věty, ze které se stane kvadratická rovnice (po dosazení funkce přímky za neznámou část osy y), se vypočítají dvě možnosti souřadnice na ose x, kde ta správná leží mezi body prvotními souřadnicemi.

```
var y;
var x1 = transitionToDraw.From.X;
var y1 = transitionToDraw.From.Y;
var x2 = transitionToDraw.To.X;
var y2 = transitionToDraw.To.Y;
```

```
var a = (y2 - y1) / (x2 - x1);
var b = y1 - a * x1;

var c2 = -((radState * radState) - (x2 * x2) - (y2 * y2) + (2 * y2 * b) - (b *
    b));
var a2 = 1 + (a * a);
var b2 = (2 * a * b) - (2 * x2) - (2 * y2 * a);

var diskriminant = (b2 * b2) - (4 * a2 * c2);

if (diskriminant > 0) {
        var reseni1 = (-b2 - Math.sqrt(diskriminant)) / (2 * a2);
        var reseni2 = (-b2 + Math.sqrt(diskriminant)) / (2 * a2);
if ((reseni1 > x2 && reseni1 < x1) || (reseni1 < x2 && reseni1 > x1)) {
        x = reseni1;
      }
      else {
        x = reseni2;
      }
}
else {
     x = (-b2 + Math.sqrt(diskriminant)) / (2 * a2);
}
y = (a * x) + b;
```

Výpis 5: Posunutí šipky na okraj stavu

Pro vysvětlení, konstanta `radState` je poloměr kolečka stavu a zde to je také délka od středu stavu odkud se má vykreslit šipka směřující do tohoto stavu. Poté se už může zavolat funkce pro vykreslení celé přímky se šipkou `drawArrow`. Aby se nemusel vypočítávat i začátek z okraje prvního stavu, vždy se přechody vykreslují na canvas dříve než stavy (tím se začátek ze středu stavu překreslí samotným stavem).

Dále je potřeba zobrazit znaky, kterými se přechodová funkce vyznačuje. A to pomocí funkce, kterou jsem popsal už při vykreslování stavu, fillText.

```
var x;
var xAbcd = (x1 - x) / 2;
xAbcd += x + abcdPositionXplus;
if (xAbcd < 0)
    xAbcd = -xAbcd;
```

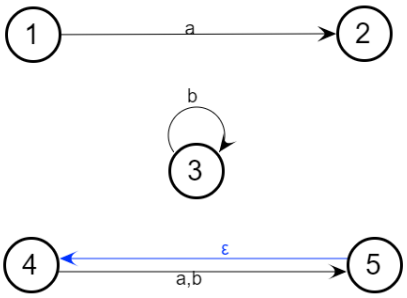
```
var yAbcd = (y1 - y) / 2;
yAbcd += y + abcdPositionYplus;
if (yAbcd < 0)
    yAbcd = -yAbcd;

context.fillStyle = collor;
context.font = "20px Arial";
context.beginPath();
context.fillText(transitionToDraw.Abcd, xAbcd, yAbcd);
context.closePath();
```

Výpis 6: Zobrazení znaků u přechodu

Text jsem musel posunout do středu přímky (přesněji podle kódu do poloviny mezi souřadnicemi stavu prvního a souřadnicemi konce šipky) plus o pozici `abcdPositionXplus` a abcdPositionYplus, která je zde kvůli přechodům v opačném směru, tak aby jeden z přechodů měl text pod přímkou a druhý nad přímkou. Tyto proměnné se vypočítávají ještě před voláním funkce pro vykreslení přechodu.

Pokud přechod směřuje ze stejného stavu jako vychází, musí se místo přímky použít křivka, která se vykresluje lehce pomocí funkce drawArcedArrow(ctx, x, y, r, startangle, endangle, anticlockwise, style, which). S touto funkcí se pracuje jako s vykreslováním kružnice, kde `x`, `y` je střed kružnice a `r` je poloměr, pak `startangle` je začátek vykreslování a *endangle* je konec. Dále parametr `anticlockwise` je jakým směrem se má křivka vykreslovat a `style` i `witch` jsou stejné parametry jako poslední dva u funkce `drawArrow`. Poté se ještě musí vypsat nad křivku znaky přechodu.

Přechod může být i v protisměru druhého přechodu. Pokud tato situace nastane, musí se jeden přechod posunout dále od přechodu druhého. Toto posunutí se řeší před voláním funkce pro vykreslení přechodu a je řešeno posunutím v ose x nebo v ose y podle toho, v jakém kvadrantu se přímka nachází, kde střed os je přesně ve středu stavu, od kterého se přímka vede. Pro ještě lepší oddělení těchto přechodů, je vždy jeden z nich vykreslený modrou barvou. Níže na Obrázku 20 jsou vyobrazeny všechny přechody, které jsem zde popsal, ve finální aplikaci.



Obrázek 21: Ukázka přechodů na výstupu canvasu

Funkce Drag And Drop

Manipulace s celým grafem, poté probíhá pomocí dolních tlačítek v první řadě, kde může být aktivované (pokud je aktivní, je zabarveno v jiné barvě než ostatní) jenom jedno tlačítko. Pokud není aktivované ani jedno tlačítko můžou se stavy aspoň přesunovat pomocí tzv. drag and drop. K této funkčnosti mi pomohl tutoriál na stránkách Rectangleworld [6]. Funkce se skládá z více kroků a začíná se odchycením události stlačení levého tlačítka myši (`mousedown`) na canvasu. V této události se najde pomocí funkce `hitTest` nad jakým objektem (stavem) se ukazatel myši nachází a tento se nastaví jako aktuální posouvaný objekt.

```
function hitTest(shape, mx, my) {
    var dx;
    var dy;
    if (shape == null)
        return false;
    dx = mx - shape.X;
    dy = my - shape.Y;
    return (dx * dx + dy * dy < shape.Rad * shape.Rad);
}
```

Výpis 7: Funkce hitTest

V posledním řádku kódu jde vidět, že vzdálenost od kruhového objektu nemůže být větší než jeho poloměr. Dále se nastaví odchycení nové událost `touchmove`, starající se o přesné posouvání objektu ve směru pohybu myši. Zde se musí dávat pozor na objekt, jenž je tady brán jako kolečko a jeho souřadnice jsou ve středu, aby nepřesáhl okraj canvasu.

V události `mousedown` se také nastavuje odchycení události uvolnění tlačítka na myši (`mouseup`), které ukončí posouvání objektu tím, že odhlásí odchytávání události `touchmove`.

Drag And Drop funkce se dále používá při vkládání nového přechodu, kdy začátek (drag) posouvání je na přechodu, od kterého se povede přechodová funkce a konec (drop) na stavu, kde přechod směřuje. Zde při této funkci se vykresluje šipka vedená od stavu k ukazateli myši

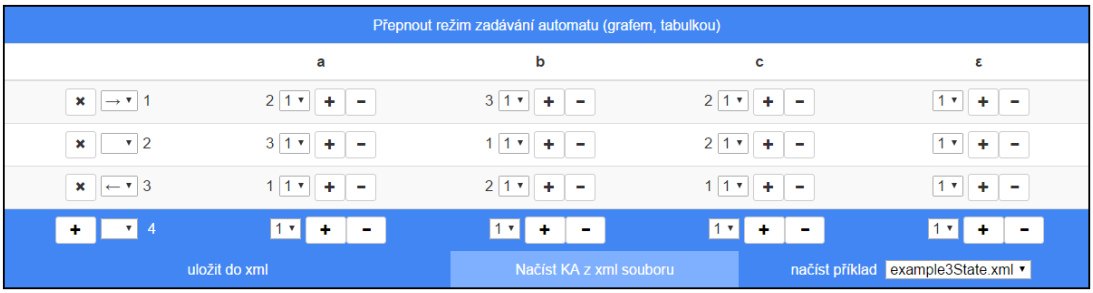
(nemusí se vypočítávat konec šipky jako u normálního přechodu) a v události `mouseup` se zjistí, na jakém stavu ukazatel skončil a vloží se nový přechod do automatu.

5.2.3 Automat zobrazený přechodovou tabulkou

Další přístup vložení automatu uživatelem, je přechodovou tabulkou. Mezi těmito režimy vkládání se dá přepínat pomocí tlačítka na horní části komponenty pro vstup a výstup automatu, kdy se po kliknutí, jeden přístup vkládání schová a druhý zobrazí.

Přechodová tabulka se zde zobrazuje pomocí HTML elemtu table, kde první řádek, stejně jako normální přechodová tabulka, je pro znaky abecedy automatu, další řádky odpovídají jednotlivým stavům a poslední řádek označený modrou barvou je pro vložení nového stavu. Pro vložení nového stavu se vždy používá číslo následující od stavu s největším číslem. Na řádku pro vložení nového stavu je dále na výběr přímo vedle čísla stavu tzv. dropdownlist (v HTML element select) pro výběr typu stavu (v modelu pod `Type`) a pod sloupci označenými znaky abecedy jsou rovnou zobrazeny vstupy pro vložení přechodů směřující z tohoto stavu. Vstupy pro vložení nového přechodu se skládají z dalšího dropdownlistu, který v sobě má čísla všech stavů, jenž jsou momentálně v automatu vloženy, dále tlačítko se znakem plus pro přidání toho přechodu a tlačítko se znakem mínus pro odebrání přechodu (pokud je automat nedeterministický a zároveň má v tomto stavu jedním znakem více přechodů, bere se pro smazání vždy poslední přidaný přechod). Typ stavu se také může měnit přímo u vloženého stavu dropdownlistem stejně jako u stavu, který se přidává. Zde se také musí dávat pozor, na deterministický automat, který nemůže mít víc jak jeden přechod jedním znakem z jednoho stavu a také nemůže mít víc jak jeden počáteční stav. Proto se tlačítka pro přidávání nového přechodu vypínají, pokud už stav nějaký přechod pomocí daného znaku má a pokud už automat má nadefinovaný počáteční stav odebere se možnost v dropdownlistu změny jiného stavu na počáteční.

Celé vytváření tabulky nad automatem se dělá samozřejmě pomocí Javascriptu a to funkcemi `createElement` nad objectem document, kde se jako parametr volí jméno HTML elementu, dále například funkcemi jako `appendChild`, která přidává k elementu jeho potomky (například u elementu table je potomek řádek tr), nebo hodně využívána funkce `setAttribute`, která přidá nebo změní atribut HTML elemtu. Celý proces vytváření této tabulky začíná prvním řádkem, který se vytváří procházením abecedy automatu, poté procházením každého stavu automatu, kde se nejenom vytvoří řádek stavu, ale také vypíšou jeho přechody. Na konci se ještě vytvoří poslední řádek pro vložení nového stavu. Výsledný automat poté vypadá jako na obrázku níže (Obrázek 21).



Obrázek 22: Automat zobrazen pomocí přechodové tabulky v aplikaci

5.2.4 XML soubor automatu

Pro uložení automatu do souboru, aby se poté mohl znovu nahrát jako vstup, jsem zvolil formát XML, kvůli jeho jednoduchého čtení jak strojově tak člověkem. Při ukládání se automat z prohlížeče klienta pošle pomocí AJAXu, ve formátu JSON (model pro automat je popsaný výše v kapitole 1.2.1), na server (přesněji pomocí funkce `GetXmlFromKA` na adrese **~api/KA/GetXmlFromKA**), kde se z něho vytvoří XML soubor a ten se jako odpověď pošle klientovi. Naopak pro načtení automatu ze souboru se pošle na API celý XML soubor a vrátí se automat v JSON formátu (tato funkce se nachází na adrese **~api/KA/GetKAFromXml**). Po načtení automatu ze serveru se musí automat vykreslit na dané komponentě automatu funkcí `drawScreen`, která zavolá jak funkci pro vykreslení automatu grafem, tak i funkci pro přechodovou tabulku.

Přesněji pak XML soubor popíšu na následujícím příkladu

```
<?xml version="1.0" encoding="UTF-8"?>
<ka>
  <Q>
    <state x="322" y="400">1</state>
    <state x="777" y="151">2</state>
    <state x="782" y="528">3</state>
  </Q>
  <Σ>
    <character>a</character>
    <character>b</character>
    <character>c</character>
    <character>ε</character>
  </Σ>
  <σ>
    <transition from="2" character="a" to="3"/>
    <transition from="2" character="b" to="1"/>
    <transition from="3" character="b" to="2"/>
```

```
    <transition from="2" character="c" to="2"/>
    <transition from="1" character="a,c" to="2"/>
  </σ>
  <σ0>
    <start>1</start>
  </σ0>
  <F>
    <end>3</end>
  </F>
</ka>
```

Výpis 8: Příklad zápisu automatu v XML souboru

Pojmenování elementů jsem volil, tak aby aspoň trochu odpovídaly teoretickému zápisu automatu. Pro uživatele (studenta) je lehčí se v takovém zápise automatu orientovat. Naopak nevýhoda plyne z rozdílu oproti modelu automatu, kde se musí trochu složitěji automat mapovat do daného objektu.

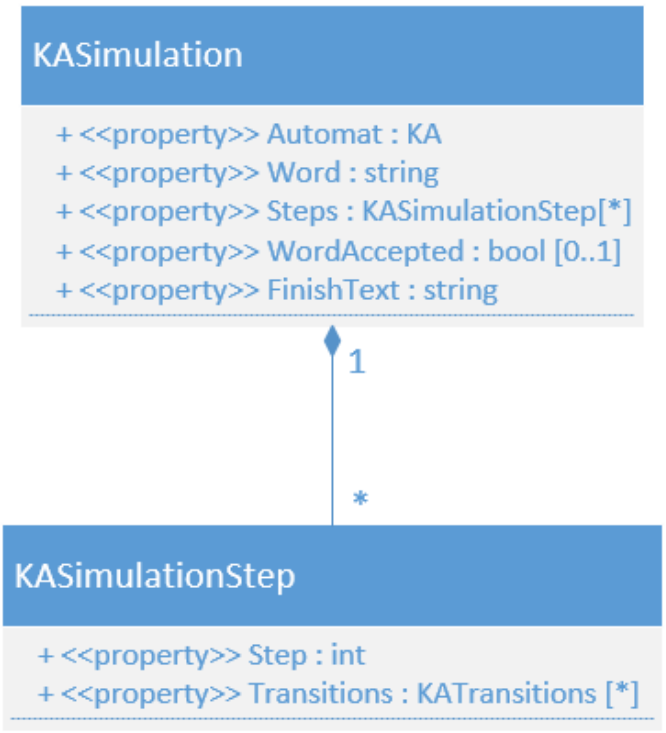
Element `Q` v sobě má stavy pojmenované číslem ve vnitřním textu a volitelné atributy `x` a `y` pro udržení pozice na HTML5 canvasu (pokud tyto atributy nejsou uvedeny, stavy budou poté rozmístěny náhodně). Element Σ obsahuje elementy character, se znaky abecedy automatu. Element s označením σ obsahuje přechodové funkce s povinnými atributy `from`, obsahující číslo stavu začátku přechodu, `to`, obsahující číslo stavu konce přechodu a konečně `character`, ve kterém jsou znaky přechodu oddělené čárkou. Dále je tu element σ`0` obsahující čísla počátečních stavů v elementech `start`. A poslední je element `F`, ve kterém jsou označeny číslem stavy přijímající v elementech `end`.

Pro uložení automatu, musí automat obsahovat minimálně aspoň jeden stav. Automat také pro načtení ze souboru, nesmí obsahovat v atributech `from` a `to` v elementu `transition` čísla stavů, které nejsou obsaženy v automatu. Taktéž nesmí být v `start` a `end` elementech jiná čísla stavu než automat má. Dále v atributech `character` se nesmí vyskytnout znak, který není v elementech `character`.

Na Obrázku 21 jde vidět rozmístění tlačítek pro tyto funkce (načtení automatu z xml souboru a uložení automatu do souboru), dále se dá všimnout ještě tlačítka pro načtení příkladu s dropdownlistem. Tato funkce automaticky načte do komponenty automatu příklad konečného automatu uloženého na serveru ve složkách, uložených ve složce App_Data. Pro každý algoritmus je zde jedna složka, popsána jménem tohoto algoritmu, obsahující XML soubory, s příklady a formátem popsaným výše. Jména XML souborů v těchto složkách jsou čteny při vytváření stránky a načteny do dropdownlistu, tak aby si uživatel mohl načíst příklad, jaký chce. Tím se zajistí, že se můžou do budoucna příklady přidávat i odebírat. Jediné pravidlo zde je, aby soubor měl příponu `.xml`.

5.3 Simulování běhu automatu na daném slově

Na stránku, pro výuku hlavní funkčnosti konečných automatů, simulování běhu automatu na daném slově, se uživatel dostane přes položku **Simulování běhu** v menu aplikace. Tato stránka se skládá ze vstupu (HTML input) pro slovo, nad kterým simulace bude probíhat. Tento vstup je scriptem kontrolován, aby se nemohlo zadat jiné slovo než ze znaků abecedy automatu. Dále je zde tlačítko na spuštění simulace a hlavní komponenta pro vstup automatu.



Obrázek 23: Model KASimulationi

Hlavní třídou je `KASimulation`, kde vlastnost `Automat` má v sobě celý automat nad kterým je slovo simulováno. `Word`, jak už z názvu napovídá, je dané slovo nad abecedou automatu vložené uživatelem. Dále vlastnost s typem nullable bool `WordAccepted` značí konec simulování a také pokud slovo bylo přijato nebo ne. Na konci simulování je také vypsaný `FinishText`, ve kterém je popsáno proč bylo slovo přijato, popřípadě nebylo. V poslední vlastnosti `Steps` jsou přímo popsané kroky, které simulace už udělala. Tato vlastnost je zde hlavně proto, aby si simulace dokázala zapamatovat předchozí kroky a jednoduše se k nim mohla vrátit (později tlačítkem **Zpět**).

Ve třídě `KASimulationStep` pak stačí mít vlastnost `Step`, udávající číslo kroku, tak aby se dali kroky seřadit od prvního do posledního, a `Transitions` jsou přechody, které byli v daném kroku použity (pokud je automat deterministický bude vždy v `Transitions` jenom jeden přechod).

Začátek simulace se provádí stiskem tlačítka **Start**, kde se volá na server funkce `Simulation-Start` přesněji na adrese **~/api/RunSimulationApi/SimulationStart**, která jako vstup má model `KASimulation` a vrací také `KASimulation`. Na začátku této funkce, se zkontrolují vstupy, přesněji musí mít automat aspoň jeden stav. Poté se najdou všechny stavy, které jsou počáteční (`KAState.Type` je Start nebo StartEnd), a vloží se do samostatné kolekce, dále zkontroluje epsilon přechody z počátečních a přidá jejich stavy, do kterých míří, do této kolekce. Dále všechny tyto stavy obarví zelenou barvou, aby šlo vidět na grafu, kde simulace začíná a vytvoří první objekt `KASimulationStep`, s krokem číslo jedna, do kterého vloží `KATransition` objekty jenom s jednou vyplněnou vlastností `To`, ve které budou stavy z kolekce. U klienta se začátek projeví obarvením počátečních stavů a k nim epsilon přechodů na zeleno, dále se objeví tabulka s vyznačením, na jakém znaku se simulace nachází, zmizí tlačítko **Start** a objeví se tlačítka s nápisem **Další** a **Zpět**. Nakonec se ještě vypnou vstupy pro manipulaci s automatem.

Simulace poté pokračuje tlačítkem **Další**, kde se volá funkce `SimulationNext` na adrese **~/api/RunSimulationApi/SimulationNext**, které se předává objekt třídy `KASimulation`. Prvně se obarví celý automat výchozí barvou (kvůli předešlému kroku, kdy byli některé stavy a přechody nabarveny na zelenou barvu), vybere se poslední krok, který se v simulaci provedl a podle něho se vybere další znak ve slově a stavy, od kterých se bude dále pokračovat tímto znakem. Poté se projdou přechody ze stavů, od kterých se pokračuje, a tyto přechody a stavy, ke kterým směřují, se obarví zase na zelenou barvu, udávající další krok. Pokud znak v tomto kroku je poslední ve slově, je to poslední krok a musí se vyplnit vlastnosti `WordAccepted` a `FinishText`. Jestliže automat slovo přijal se zjistí tak, že aspoň jeden stav ze stavů, které se v tomto kroku nabarvili na zeleno, je stavem přijímajícím. FinishText se pak vyplní čtyřmi možnými způsoby:

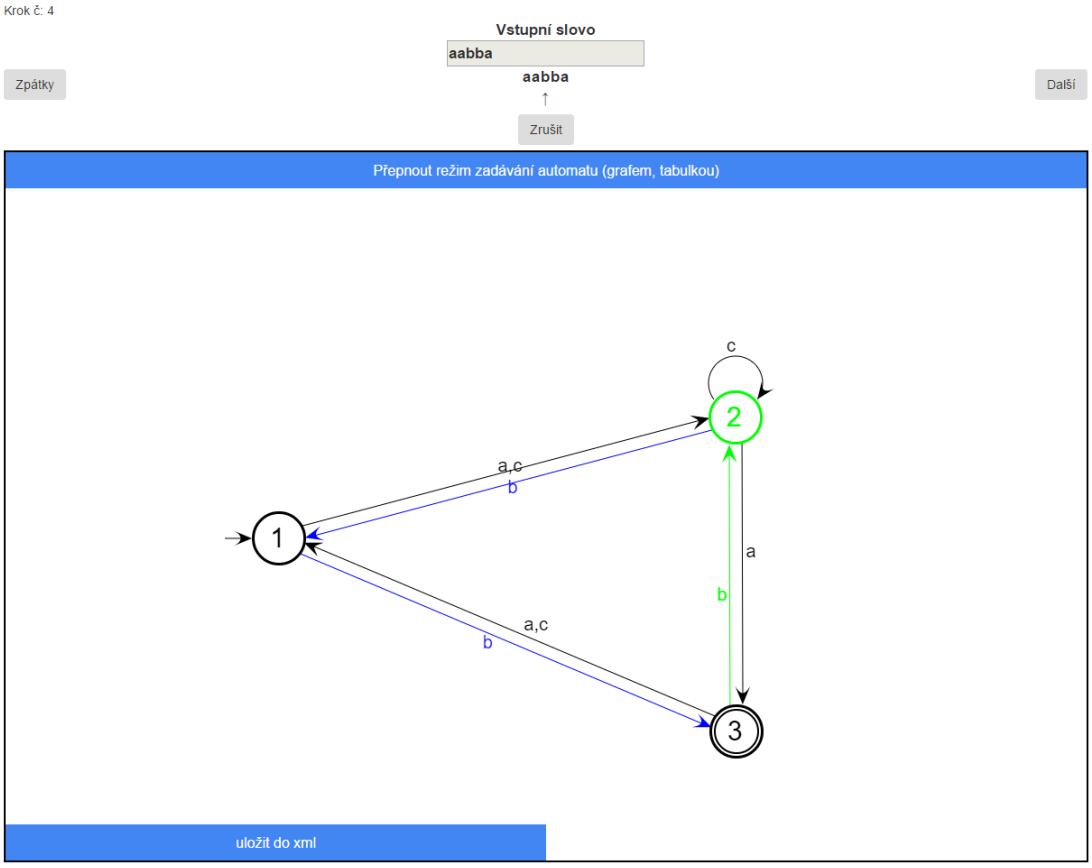
- „Slovo bylo přijato ve stavu {0}“ : pokud slovo bylo přijato a automat je DKA
- „Slovo bylo přijato, protože jeden z možných běhů skončil v přijímajícím stavu {0}": pokud slovo bylo přijato a automat je NKA
- „Slovo nebylo přijato, protože simulace neskončila v přijímajícím stavu“ : slovo nebylo přijato a automat je DKA
- „Slovo nebylo přijato, protože žádný z možných běhů neskončil v přijímajícím stavu": slovo nebylo přijato a automat je NKA

Kde {0} znázorňuje číslo stavu, nebo u NKA může těchto stavů být více oddělených čárkou. U nedeterministického automatu může také dojít k případu, kdy se nenajde žádný přechod v dalším kroku odpovídající danému znaku ze stavů z předešlého kroku. Tato situace se řeší rovnou vyplněním vlastnosti `WordAccpted` na hodnotu false a do `FinishText` se zapíše "Slovo nebylo přijato, protože přechodová funkce automatu neumožňuje přečíst celé slovo do konce". Funkce následně vrátí vyplněný objekt třídy `KASimulation`. U klientského prohlížeče se znovu překreslí automat, kvůli jiným obarveným stavům. Šipka ukazující, na jakém znaku ve slově

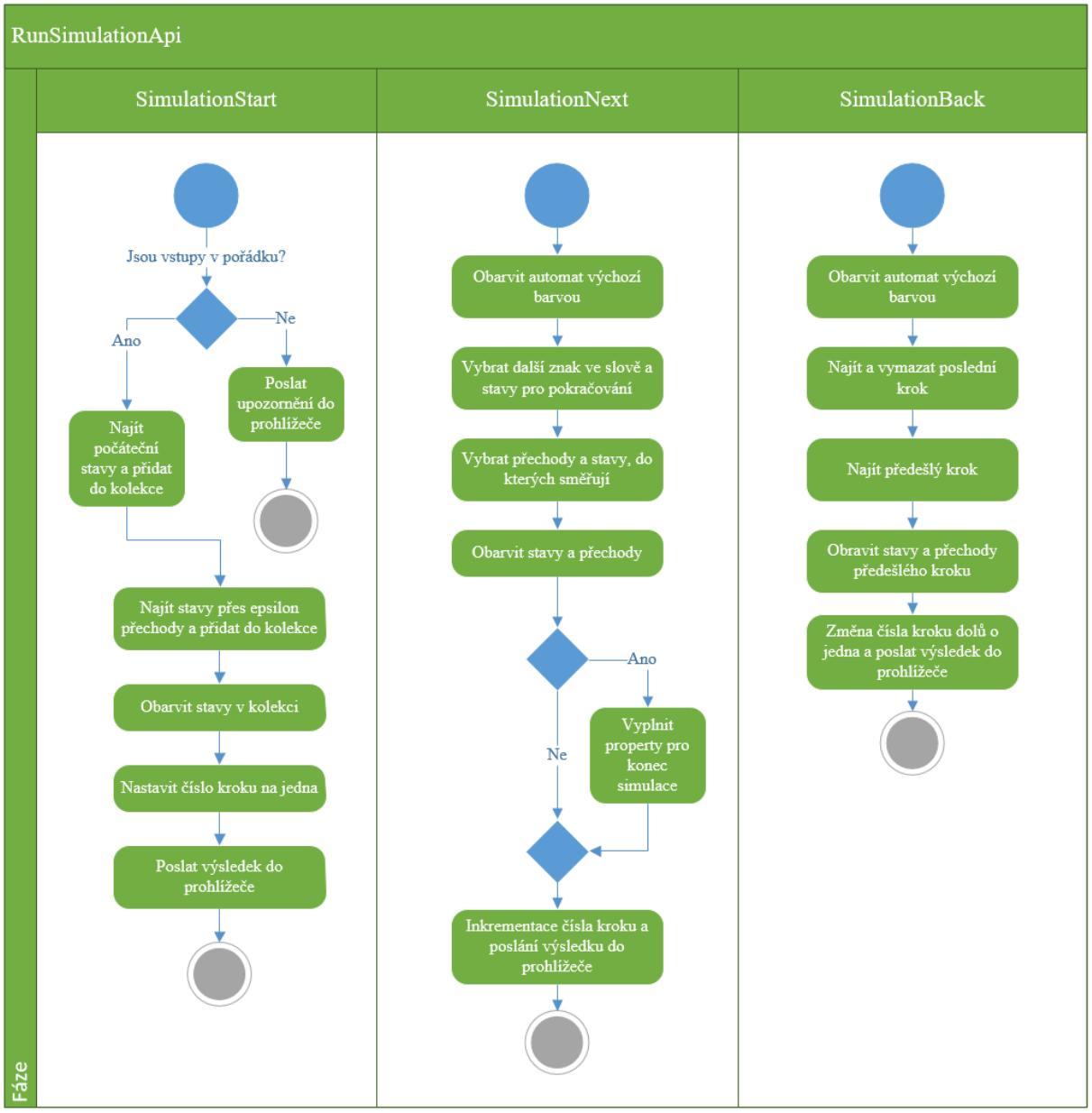
se simulace nachází, se posune o jeden znak doprava. Pokud je simulace u konce (je vyplněná hodnota v proměnné `WordAccepted`) ukáže se text v `FinishText` uživateli a tlačítko **Další** zmizí.

Po stisku tlačítka **Zpátky** se volá funkce `SimulationBack` (**/api/RunSimulationApi/SimulationBack**), které se znovu předává objekt třídy `KASimulation`. Na začátku se znovu celý automat nabarví na výchozí barvu a vybere se poslední krok, který simulace provedla. Pokud je tento krok počátečním (`KASimulationStep.Step` = 1) jednoduše se jenom vlastnost `Steps` vyplní hodnotou null, pokud tento krok naopak není počáteční, celý krok se z pole `Steps` vymaže a poté se vybere předešlý krok a podle jeho přechodů se vybarví na zeleno stavy a přechody tohoto kroku. Funkce samozřejmě vrací objekt třídy `KASimulation`. Na klinetské části se graf automatu znovu přepíše a šipka se posune o jeden znak doleva. Pokud proměnná `Steps` není vyplněná, znamená to, že simulace se má ukončit, tím se celá stránka přepne do stavu, ve kterém byla před simulací.

Při simulaci je zde také tlačítko s nápisem **Zrušit**, které pomocí javascriptu jednoduše ukončí celou simulaci a stránku přepne do stavu, ve kterém byla před simulací.



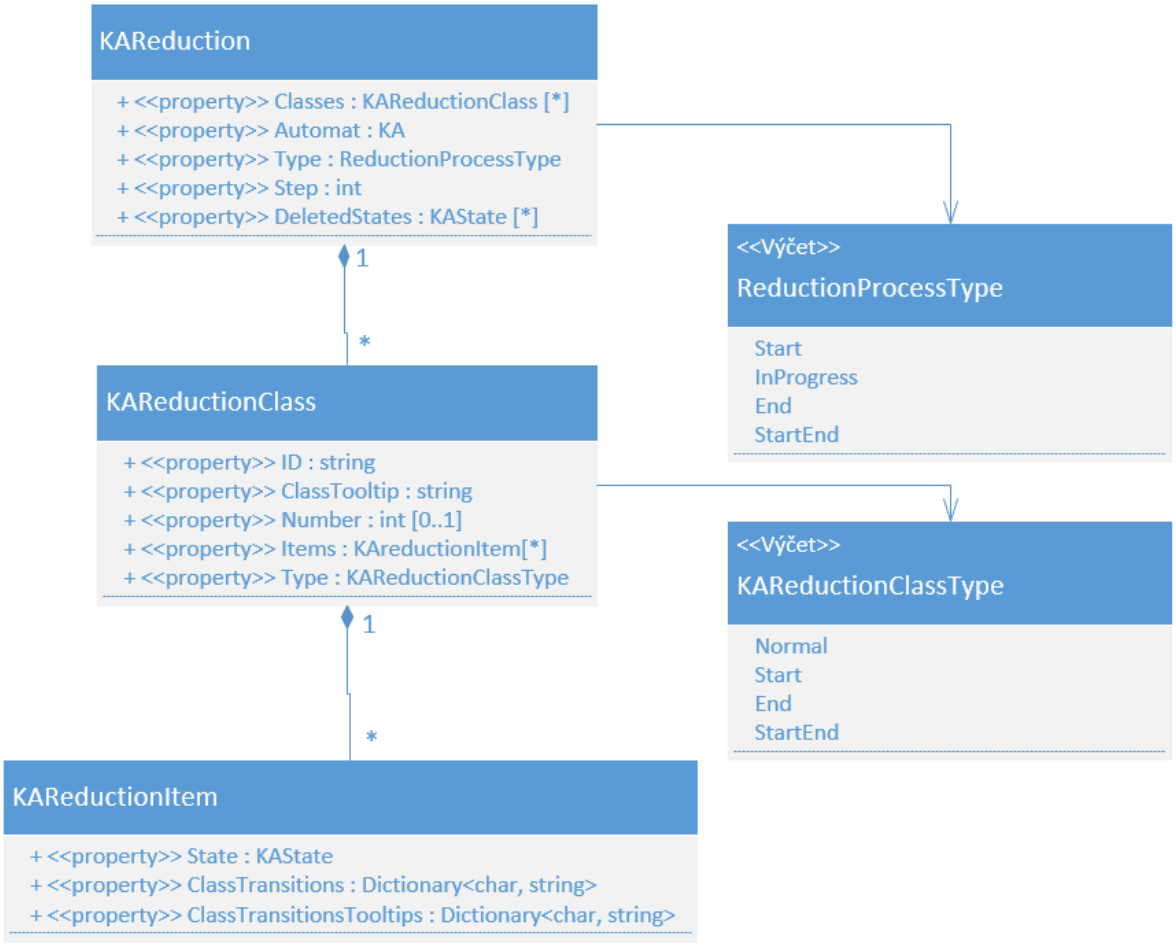
Obrázek 24: Simulace běhu na daném slově ve výsledné aplikaci



Obrázek 25: Diagram aktivit pro simulaci běhu na daném slově

5.4 Redukce deterministického automatu

Stránka pro redukci automatu na začátku redukce vypadá jednoduše a to, že obsahuje jenom komponentu pro vstup automatu (komponenta je nastavena na režim deterministického automatu) a pod ní tlačítko pro začátek redukce s nápisem `Start`. Na tuto stránku se uživatel dostane pomocí odkazu v menu s názvem **Redukce automatu**.



Obrázek 26: Model KAReduction

Začnu s hlavní třídou v diagramu `KAReduction`. Vlastnost `Automat` je zde samozřejmě jako automat, který se redukuje, ale také je zde pro automat už redukovaný. `Type` typu výčtového `ReductionProcessType` je zde aby identifikoval, jestli redukce je teprve na začátku (`Start`) nebo v tomto kroku pořád probíha (`InProgress`), dále může už být na konci (`End`), nebo začala a skončila zároveň (`StartEnd`). Další Properite `Step` je zde, stejně jako v simulaci, pro určení, kolikátý krok se provedl. `DeletedStates` se využivá jenom na začátku redukce a obsahuje stavy nedosažitelné z počátečního stavu, které můžeme hned odstranit. Hlavní vlastnost je zde `Classes`, která obsahuje přesné třídy rozkladu.

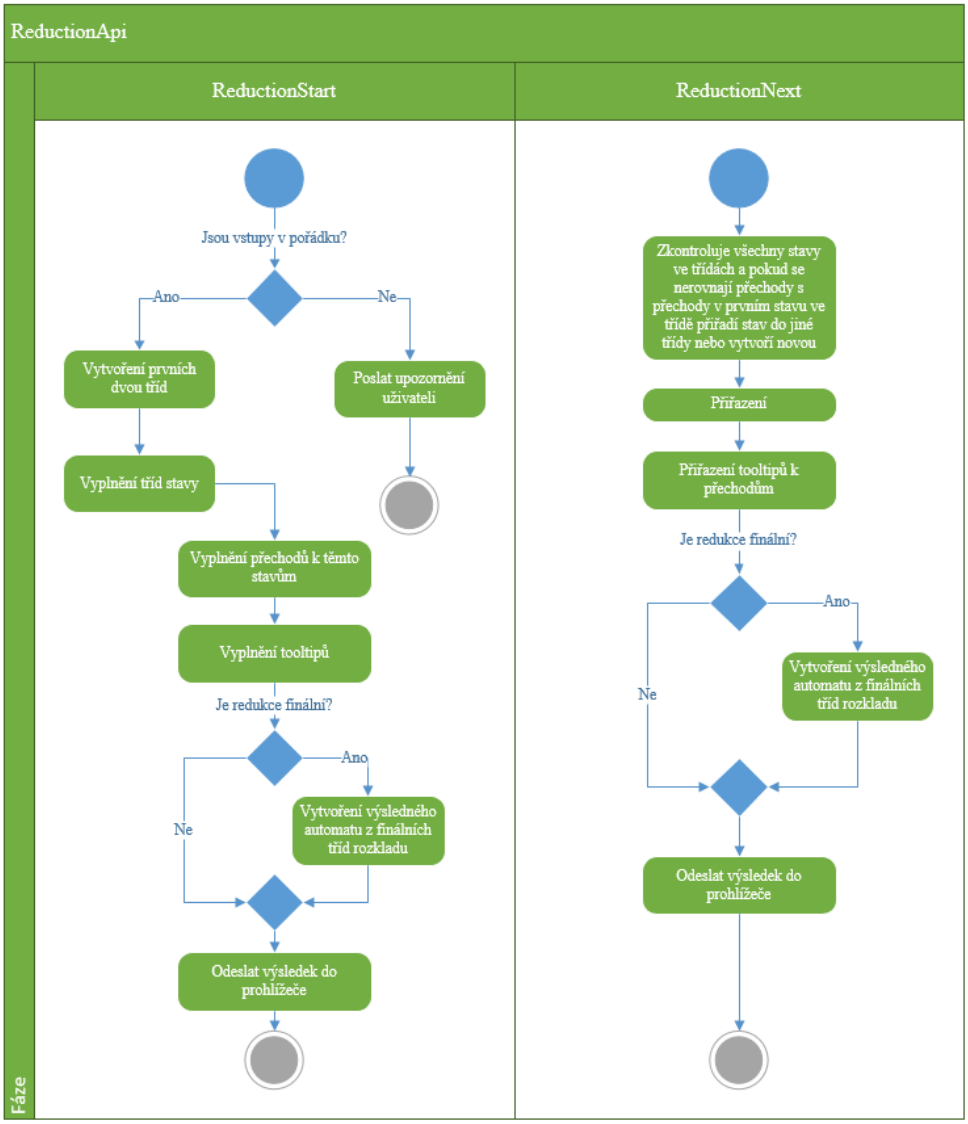
Třída rozkladu je zde reprezentována třídou `KAReductionClass`. Kde vlastnost `ID` je římské číslo třídy rozkladu, `ClassTooltip` obsahuje text, který se bude uživateli objevovat jako tooltip po najetí cursoru na číslo třídy pro pochopení, jak tato třída vznikla, ve výsledné tabulce třídy rozkladu. Dále `Number` typu nullable intiger se využívá jenom při posledním rozkladu (`KAReduction.Type` s hodnotou `End` nebo `StartEnd`) a má v sobě výsledné číslo stavu, do kterého se tato třída zobrazuje. `Type` poté nese informaci, zda třída má v sobě jenom přijímající

stavy (`End`), nebo stavy přijímající nemá (`Normal`). Dále na konci redukce může mít v sobě stav počáteční a tím se vyznačuje hodnotami `Start` nebo `StartEnd`. Konečně pole `Items` obsahuje stavy (`KAReductionItem.State`), a k nim přechody do kterých tříd se ze stavu dostane (`KAReductionItem.ClassTransitions` obsahující objekt třídy `Dictionary`, kde klíčem je znak přechodu a hodnota je římské číslo třídy do které se dostane). Je zde také v `KAReductionItem` vlastnost `ClassTransitionsTooltips`, ve které jsou stejně jako v `ClassTooltip` uloženy texty pro uživatele, aby lépe porozuměl, proč směřuje daný přechod do dané třídy.

Začátek celé redukce začíná stiskem tlačítka **Start**. V tomto okamžiku se volá funkce `ReductionStart` na adrese **~/api/ReductionApi/ReductionStart**, které stačí předat redukovaný automat a funkce na konci vrací objekt třídy `KAReduction`. Ve funkci se na začátku musí zkontrolovat vstupy. Zde se také kontroluje, zda automat je deterministický. Pokud nejsou vstupy v pořádku pošle se upozornění uživateli. Pokud naopak jsou vstupy v pořádku začíná se vytvořením dvou tříd rozkladu (`KAReductonClass`), do kterých se vloží stavy (v `KAReductionItem.State`) s přechody do těchto tříd podle přechodů normálních v počátečním automatu. K nim se přiřazuje i tooltip (`KAReductionItem.CLassTransitionsTooltips`) pro uživatele (např. „Automat ze stavu 1 přechází znakem 'b' do stavu 3, který je aktuálně ve skupině II“). První třída (římská I) rozkladu pak v sobě bude mít jenom stavy nepřijímající a je také označená `KAReductionClass.Type` hodnotou Normal. Ještě musí v `KAReductionClass.ClassTooltip` obsahovat text „Skupina I obsahuje nepřijímající stavy“. Druhá třída (římská II) obsahuje stavy přijímající a je označena v `KAReductionClass.Type` jako End. V `ClassTooltip` má poté text „Skupina II obsahuje přijímací stavy“. Dále algoritmus musí rovnou zkontrolovat, zda není redukce ve finální třídě rozkladu. Jednoduše prochází všechny stavy ve všech třídách a pokud se nerovná aspoň jeden přechod ve stavu s přechody v prvním stavu ve třídě, redukce není finální. Pokud ale je redukce finální, musí se z těchto dvou tříd vytvořit nový automat a ten pak předat v odpovědi pomocí vlastnosti `Automat`. Také se ve finální redukci vyplňuje vlastnost `KAReductionClass.Number` číslem stavu, který ve výsledném automatu reprezentuje tuto třídu. Na straně klienta poté zmizí tlačítko **Start** a místo něho se vytvoří z vypočítaných tříd tabulka tříd rozkladu obsahující jak třídy, tak k nim přiřazené stavy s přechody, do jakých tříd se pomocí znaku stav dostane (tato tabulka je popsaná v kapitole Konečné automatu a podkapitole Redukce automatu). K označeným třídám v tabulce se pomocí bootstrap tooltip pluginu přiřadí hodnoty z `KAReductionClass.ClassTooltip` aby uživatel po najetí kurzoru nad třídou viděl pop-up text, vysvětlující, jak se třída vytvořila. To samé se vytvoří i u přechodů. Pokud redukce není finální (není označena jako End, StartEnd) přidá se za tabulku tlačítko pro další krok, se šipkou směřující dolů. Pokud ale je redukce finální objeví se nová komponenta automatu a do ní se vloží výsledný automat.

Při dalším kroku se už volá funkce `ReductionNext` na adrese **~/api/ReductionApi/ReductionNext**. Do funkce vstupuje objekt třídy `KAReductionClass` z předešlého kroku. Zde algoritmus pokračuje tím, že prochází předchozí vytvořené třídy s tím, že kontroluje v jedné třídě přechody do tříd s prvním stavem ve třídě a pokud se aspoň jeden přechod nerovná, vezme

tento stav a vloží ho do třídy obsahující stejné přechody. Pokud tato třída neexistuje, vytvoří ji, stav do ní rovnou vloží a přiřadí ji `ClassTooltip` (např. „Skupina III obsahuje ty stavy ze skupiny I, které měly přechody přes znaky a,b do skupiny I,I.“), podle rozdílu oproti předchozí třídě. Dále se procházejí znovu všechny stavy (`KAReductionItem`) ve všech třídách a přiřazují se jim tooltipy (stejné jako na začátku redukce ve funkci `ReductionStart`) k přechodům. Ke konci se znovu zkontroluje, jestli není redukce finální, pokud ano vytvoří se z ní automat stejně jako ve funkci `ReductionStart`. Než se pošle výsledek výpočtu klientovi inkrementuje se číslo kroku. U klienta se znovu vytvoří nová tabulka tříd rozkladu a pokud je redukce finální, zobrazí se finální konečný automat v komponentě automatu.



Obrázek 27: Diagram aktivit pro redukci deterministického automatu

Krok č. 1

		a	*b*
I	1	I	II
	2	I	II
	4	I	I
	5	I	I
II	3	II	I
	6	II	I

Krok č. 2

		a	*b*
I	1	I	II
	2	I	II
II	3	II	III
	6	II	I
III	4	I	III
	5	I	III

Obrázek 28: Příklad výsledné tabulky tříd rozkladu

5.5 Převod nedeterministického konečného automatu na deterministický

Stránka převodu vypadá na začátku převodu stejně jako předchozí redukce deterministického konečného automatu a nachází se v menu pod názvem zkráceně **NKA na DKA**, kvůli ušetření místa v menu aplikace.

Obrázek 29: Model KAConvertNKA

Tento model začíná vlastnosti `AutomatNKA` obsahující celý nedeterministický automat, který

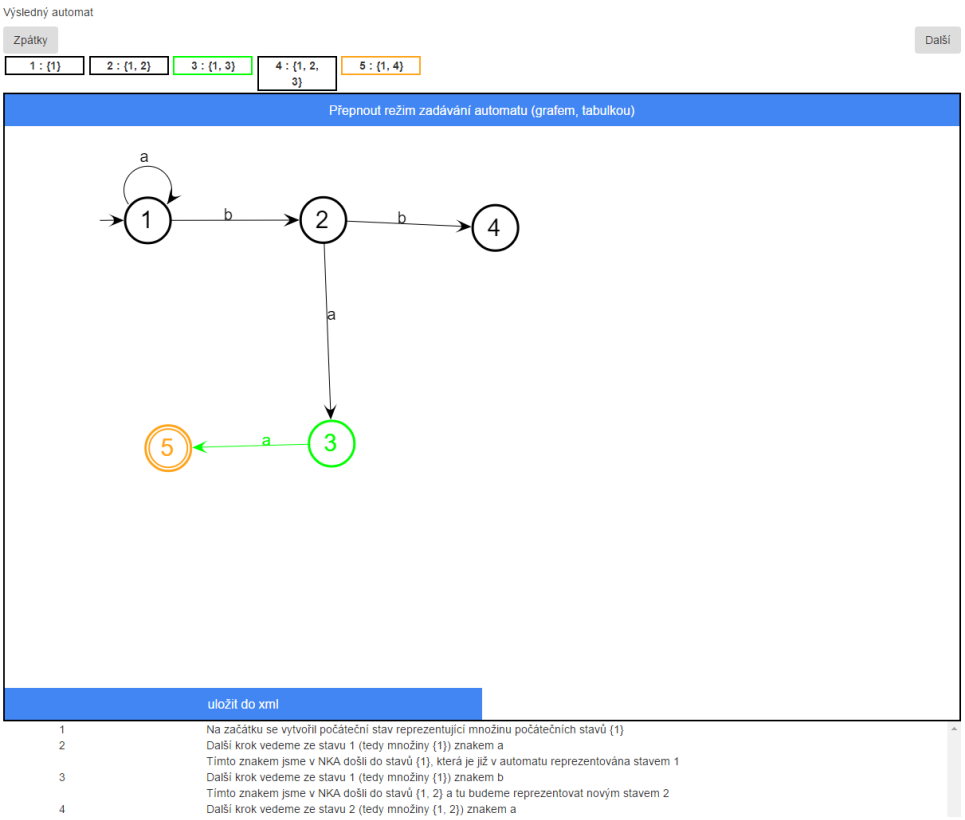
se bude převádět na deterministický a ten je poté uložený v další vlastnosti `AutomatDKA`. `End` je poté ukazatel, jestli převod automatu není u konce. `Step` pak samozřejmě udává číslo kroku a kolekce `StepComments` obsahuje seřazeně po kroku komentáře upřesňující pro uživatele co se daný krok provedlo. Poslední je `StateConvertor` sloužící pro převod čísla stavu vytvořeného v DKA na skupinu stavů v NKA, protože výsledné stavy automatu jsou také číslovány od jedničky nahoru a uživatel poté potřebuje vědět z jakých stavů NKA se stav v DKA skládá. Tato vlastnost typu `Dictionary`, kde jako klíč je číslo stavu v DKA a jako hodnota je zde kolekce čísel představující stavy v NKA.

Uživatel převod začíná, tak jako u každého algoritmu vložením automatu do komponenty a následným stisknutím tlačítka **Start**, kdy dojde k odeslání automatu na server do funkce `ConvertNKAStart` na adrese **~/api/ConvertNKAApi/ConvertNKAStart** (automat se posílá rovnou v modelu `KAConvertKA`). Na začátku se zkontrolují vstupy, kde se kontroluje, jenom jestli nějaký automat na vstupu vůbec je (automat nesmí být prázdný). Pokračuje se vybráním počátečních stavů ze stavů NKA a k nim se přiřadí stavy do kterých se dojde pomocí epsilon přechodů z počátečních stavů. Tyto stavy a epsilon přechody se nabarví na zelenou barvu a vytvoří se z nich ve výsledném DKA automatu první stav (stav číslo jedna). Poté se musí ještě přidat tento stav jako klíč do vlastnosti `StateConvertor` a jako hodnota tohoto klíče se uvede kolekce stavů obarvených na zeleno (předchozí vybrané stavy). Dále se do vlastnosti `Step` přiřadí číslo kroku, kde na začátku se začíná na čísle jedna. V prohlížeči se po přijetí modelu ze serveru vstupy na komponentě automatu vypnou a zobrazí se tabulka se stavy popisující převod stavu výsledného automatu na stavy ve vstupním automatu (zde se také stavy vyznačují barvou, kterou mají právě na sobě přiřazenou). Dále se objeví seznam s komentáři, který se každý krok také nově vytváří, komponenta výsledného automatu s prvním počátečním stavem a také tlačítko pro pokračování převodu.

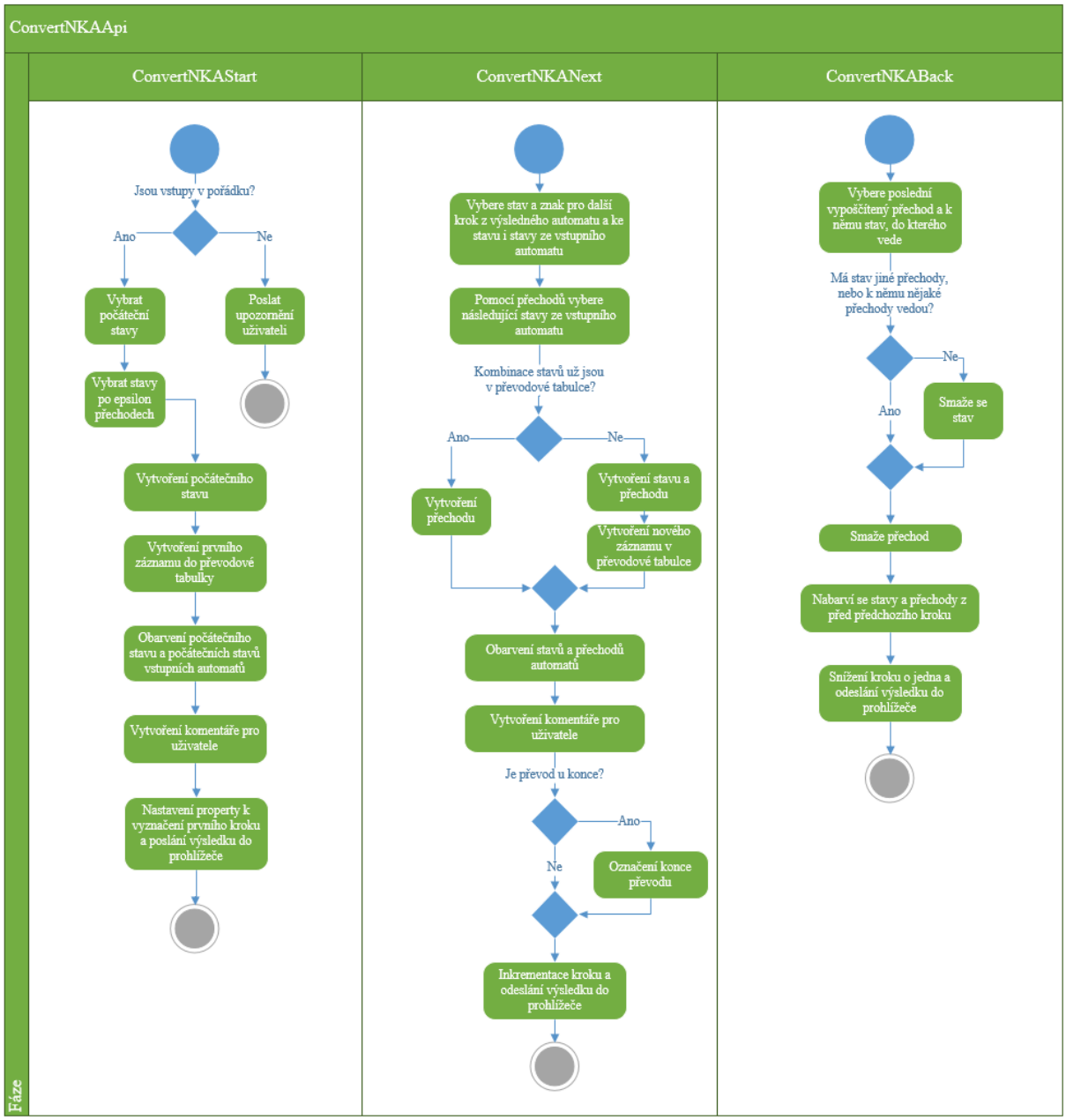
Při pokračování převodu se následně odesílá objekt převodu na server do funkce `ConvertNKA-Next` na adrese **~/api/ConvertNKAApi/ConvertNKANext**. Jako první se najde stav pro další krok ve výsledném deterministickém automatu. Ten se najde procházením seřazených stavů jednoho po druhém do té doby, než se najde stav, kterému chybí přechod nějakým znakem v abecedě. Dále se k tomuto stavu najde znak z abecedy (abeceda je zde také seřazená, kvůli tomu, aby celý postup převodu byl také seřazený), kterému algoritmus v tomto kroku vytvoří přechod z přechodů v NKA. Poté se vyberou stavy v NKA, které podle převodníku `StateConvertor` se skládají do stavu vybraného pro tento krok a obarví se na zelenou barvu. K těmto stavům se vyberou přechody, které se také přebarví na zelenou barvu, a stavy do kterých tyto přechody míří, se přebarví na oranžovou barvu. Z těchto oranžových stavů, pokud ještě nevytváří ve výsledném automatu stav, se vytvoří nový stav s číslem o jedna větší, než má stav s největším číslem ve výsledném automatu DKA a tento stav se také nabarví na oranžovou barvu. Pokud stav v DKA už existuje, jenom se nabarví na oranžovo. Při tomto převodu se každý krok vytváří komentář pro uživatele, co se v daném kroku děje a na konci se přiřadí do kolekce `StepComents` (např. „Další krok vedeme ze stavu 2 (tedy množiny {3}) znakem a“ a na dalším řádku „Tímto

znakem jsme v NKA došli do stavů {} a tu budeme reprezentovat novým stavem 3“). Na konci se kontroluje, jestli už výsledný automat není deterministický, pokud ano, do komentáře v tomto kroku se vloží text „Výsledný automat je deterministický“ a vlastnost `End` se nastaví na true. U uživatele se javascriptem poté znovu překreslí výsledný automat a převodová tabulka novými daty. Jestliže vlastnost `End` je nastavené na true, výsledný automat je deterministický a převod je u konce, a proto tlačítko pro pokračování zmizí.

Jako na stránce pro simulaci běhu automatu na daném slově, je i tady možnost vrátit se o jeden krok zpět. Pro tuto funkci je na adrese **/api/ConvertNKAApi/ConvertNKABack** funkce `ConvertNKABack`. Kde se na začátku vybere podobně jako ve funkci `ConvertNKANext` stav, který se vypočítal v minulém kroku (kroky se vypočítávají seřazeně, a proto si nepotřebujeme v modelu pamatovat všechny kroky). Vybírá se na základě čísla stavu a vypočítaných přechodů (pokud všechny stavy mají stejný počet přechodů, bere se stav s největším číslem). Poté se vybere k tomuto stavu přechod, který se v tomto procesu smaže a pokud směřoval do stavu, ke kterému nevede a ani od něho nevede žádný jiný přechod, tento stav se také smaže. Ke konci se musí ještě nabarvit stavy z před předchozího kroku, takže se celý proces opakuje s tím, že přechody ani stavy se nemažou, ale obarvují se na zelenou a oranžovou barvu. U klienta se znovu výsledný automat a tabulka přepíšou. Dále pokud krok, který se smazal, byl finální, tlačítko pro pokračování se musí uživateli objevit.



Obrázek 30: Příklad výstupu převodu NKA na DKA ve výsledné aplikace

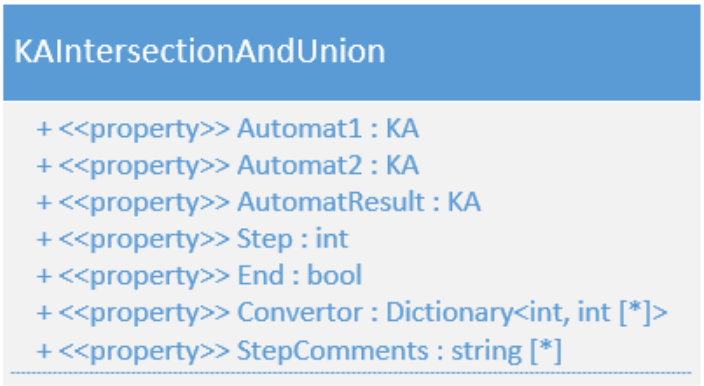


Obrázek 31: Diagram aktivit pro převod NKA na DKA

5.6 Průnik a sjednocení dvou automatů

I když jsou ve výsledné aplikaci průnik a sjednocení odděleny, tak že každý z nich má svoji stránku, budu zde popisovat oba zároveň, protože algoritmus je téměř totožný. Průnik je pak na stránce, kde se uživatel dostane pomocí menu kliknutím na **Průnik dvou automatů** a pro sjednocení **Sjednocení dvou automatů**. Jejich stránky mají v sobě před začátkem jenom dvě komponenty pro automat označeny jako „Automat č. 1“ a „Automat č.2“ a samozřejmě tlačitko **Start**.

Pro průnik i pro sjednocení se využívá stejný model `KAIntersectionAndUnion`, kdy je podobný modelu u algoritmu pro převod NKA na DKA, kvůli podobnosti v celém algoritmu. Tuto podobnost popíšu dále v textu.



Obrázek 32: Model KAIntersectionAndUnion

Rozdíl oproti modelu`KAConvertKA` je zde přidáním druhého stavu jako vstupu `Automat2` (pokud se bere `Automat1` jako `AutomatNKA` a `AutomatResult` jako `AutomatDKA`). Dále ve vlastnosti `Convertor` bude jako hodnota u klíče vždy jen dvě čísla stavu. Jinak je kromě názvu pár vlastností model stejný oproti `KAConvertNKA`.

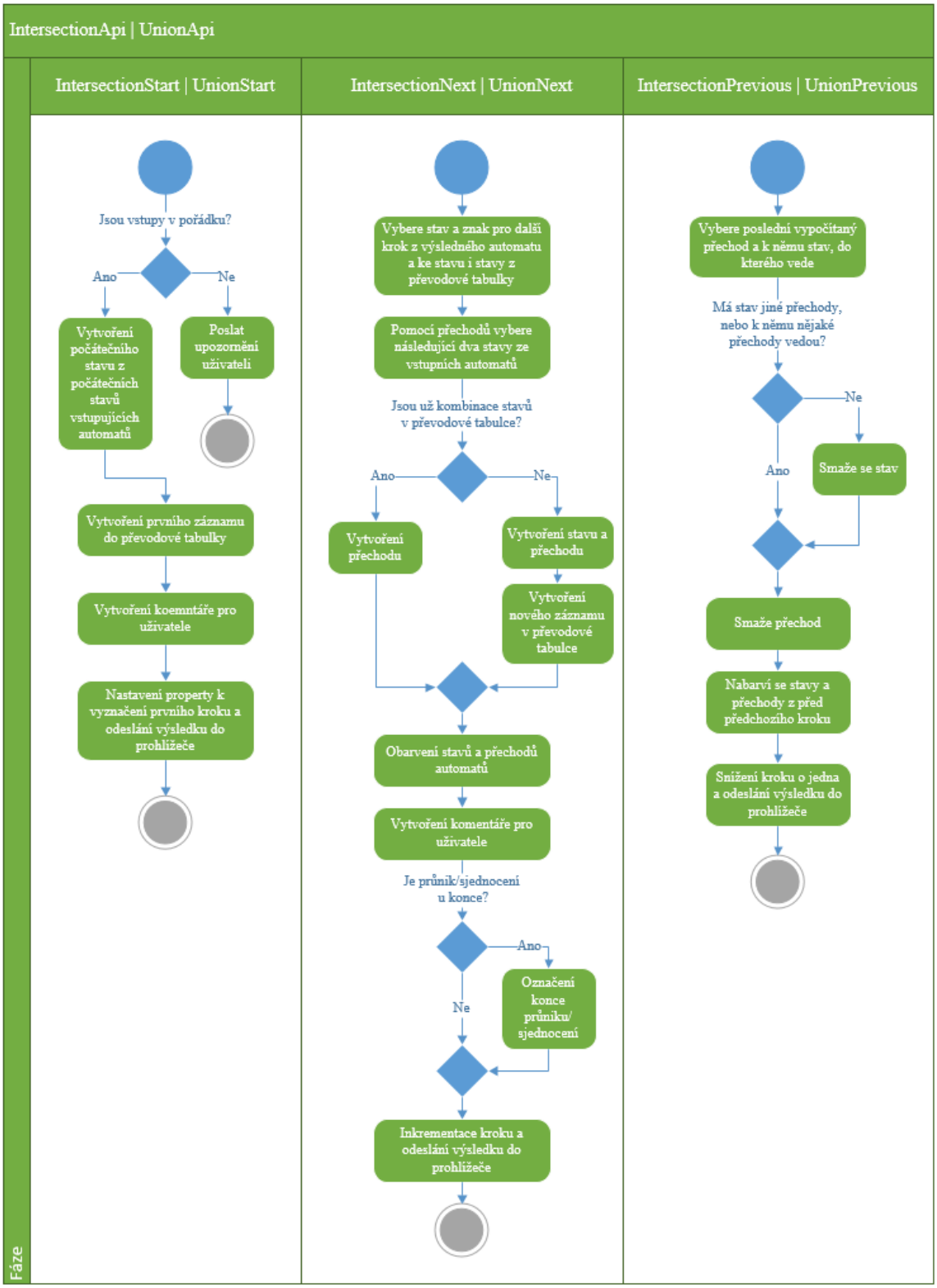
Algoritmus je podobný s algoritmem převodu NKA na DKA, protože oba prochází pomocí přechodů všechny možné stavy v automatu(ech) od počátečního a vytváří z nich automat nový, a proto mají i podobné modely.

Algoritmus začíná funkcí `IntersectionStart` (**/api/IntersectionApi/IntersectionStart**) pro průnik a pro sjednocení funcí `UnionStart` (**/api/UnionApi/UnionStart**). I když jsem i tyto funkce oddělil od sebe, jsou naimplementovány skoro stejně (o rozdílu se zmíním níže). Vstupy se kontrolují na prázdné automaty a determinismus obou vstupujících automatů. Pokud jeden z požadavků nesplňuje, odešle se uživateli přesné upozornění. Na začátku se vytvoří nový počáteční stav ze stavů počátečních prvního a druhého vstupujícího automatu (zde se nemusí kontrolovat epsilon přechody, protože oba automaty musí být deterministické), pokud se jedná o průnik bude nový stav přijímající jen tehdy, když oba dva stavy ze vstupujících automatů jsou přijímající, a při sjednocení stačí, aby aspoň jeden z nich byl přijímající (jediný rozdíl mezi algoritmem ve sjednocení oproti průniku). Dále se vytvoří `Convertor` a vloží se do něho první hodnota složená z čísla právě vytvořeného stavu a dvojicí stavů, ze kterých se vytvořila. Poté se ještě vytvoří objekt pro komentáře a přidá se první komentář pro začátek (např. „Na začátku se vytvořil stav z počátečních stavů(1, 1), tento stav je přijímající, protože oba dva stavy jsou přijímající“) a vlastnosti `Step` se přiřadí jednička. U klienta se provede to stejné jako u převodu NKA na DKA s tím rozdílem, že se výsledné stavy skládají vždy z uspořádané dvojice stavů z vstupujících automatů a proto stavy nejsou v převodové tabulce ve složených závorkách, ale v závorkách normálních.

Pro další kroky se volá funkce pro sjednocení `UnionNext` (**/api/UnionApi/UnionNext**) a pro průnik `IntersectionNext` (**/api/IntersectionApi/IntersectionNext**). Zde se stejně jako v algoritmu pro převod NKA na DKA vybírá stav a znak převodu pro následující krok. K tomuto stavu najde stavy ze vstupujících automatů a pomocí jejich převodů daným znakem dojde do další dvojice automatů, ze kterých vytvoří nový stav (jestli je přijímající se rozhoduje stejně jako ve funkci `UnionStart` nebo `IntersectionStart`), tedy pokud ještě v kolekci `Convertor` tato dvojice stavů nemá stav ve výsledném automatu, a nový přechod mezi stavem vybraným pro tento krok a stavem do něhož se algoritmus pomocí znaku dostal. Všechny tyto stavy a přechody se také obarví barvou zelenou a oranžovou pro lepší grafické znázornění tohoto kroku. Zde se také vytvářejí komentáře pro uživatele, pro lepší pochopeni algoritmu. Na konec této funkce se ještě zkontroluje, zda algoritmus není na konci (výsledný automat je u konce, pokud je deterministický). Pokud ano vlastnost `End` se nastaví na true a přidá se konečný komentář „automat je hotový, protože jsme prošli všechny stavy a přechody“. Před odesláním vypočítaného kroku do uživatelského prohlížeče se číslo kroku v `Step` inkrementuje o jedna. V prohlížeči se pak všechny automaty znovu překreslí (kvůli obarvení stavů a přechodů), přepíše se převodová tabulka a komentáře pod výsledným automatem.

Je zde také možnost vrátit se o krok zpět v algoritmu pomocí funkce u sjednocení `UnionPrevious` (**/api/UnionApi/UnionPrevious**) a průnik (**/api/IntersectionApi/IntersectionPrevious**). V těchto funkcích se prvně vybere poslední zpracovaný stav s přechodem, kde se přechod odstraní a pokud stav do kterého mířil nemá žádné jiné přechody, které k němu nebo od něho směřovaly, odstraní se také (zde se musí odstranit jak z výsledného automatu, tak z přechodové tabulky `Convertor`). Poté se celý proces znovu opakuje, ale žádný přechod nebo stav se neodstraňuje, místo toho se všechny tyto stavy a přechody v kroku předešlém obarví na zelenou nebo oranžovou barvu. Pokud je sjednocení nebo průnik na začátku (obsahuje jenom jeden počáteční stav a nemá žádný přechod) obarví se počáteční stavy na zelenou barvu. Dále se už jenom odstraní poslední komentář a krok se sníží o jedničku. U klienta se znovu provádí to stejné jako v převodu NKA na DKA po funkci `ConvertNKABack`.

Obrázek 33: Příklad výsledné převodové tabulky u sjednocení pro porovnání oproti převodu NKA na DKA



Obrázek 34: Diagram aktivit pro průnik a sjednocení dvou automatů

5.7 Zrcadlový obraz automatu

Tento algoritmus je na stránce v menu přímo pod názvem **Zrcadlový obraz automatu** a skládá se na začátku z komponenty pro vstup automatu a pod ní je šipka směřující dolů pro začátek algoritmu. Celý algoritmus je rozdělený jenom do dvou kroků a vlastní model ani mít nemusí, protože mu stačí model automatu.

Jak bylo napsáno výše tento algoritmus se skládá ze dvou kroků a z toho první krok využívá funkce `Step1` na adrese **~/api/MirrorApi/Step1**, kde předává se jenom vstupní automat. Zde se zkontroluje, jestli automat není prázdný, pokud ano pošle se uživateli upozornění. Dále se pokračuje jednoduchým procházením všech přechodů, kterým se přehodí stavy přechodu směřující od a směřující do (`From` a `To`). Poté se automat vrátí uživateli. Na straně uživatele se poté objeví komentář s textem „Změníme směr přechodů v automatu“, další komponenta pro příchozí automat a další šipka směřující dolů pro další krok.

V druhém a také posledním kroku se volá funkce `Step2` (**~/api/MirrorApi/Step2**), ve které se jenom typy stavů (`KAState.Type`) přepíšou na přijímající (End), pokud byli počáteční (Start), a nebo na počáteční, pokud byli přijímající. Výsledný automat se znovu pošle zpátky do prohlížeče. Zde se dole pod poslední přidanou šipkou objeví komentář „Přijímající stavy změníme na počáteční stavy a předchozí počáteční stavy změníme na přijímající“ a další komponenta automatu s výsledným automatem.

5.8 Doplněk

Poslední a také nejjednodušší algoritmus je na stránce, která má odkaz v menu pod **Doplněk**. Tento algoritmus, stejně jako předchozí zrcadlový obraz nemá žádný model, ale vystačí si jenom s modelem automatu. Na stránce se nachází jenom komponenta pro vstupní deterministický automat a tlačítko se šipkou směřující dolů na začátek výpočtu algoritmu. Celý proces je zde složen jenom z jednoho kroku.

Po stisknutí tlačítka se šipkou se volá funkce na serveru `Step1` na adrese **~/api/SupplementApi/Step1**, které se předává vstupní automat. Tento automat se zkontroluje, zda není prázdný a zda je deterministický, pokud ne, pošle se do prohlížeče přesné upozornění. Samotný algoritmus poté jenom projde všechny stavy automatu a přepíše jejich typy z nepřijímajícího na přijímající a naopak. Výsledek se pošle do prohlížeče, kde se objeví komentář s textem „Přijímající stavy změníme na stavy nepřijímající a stavy nepřijímající na přijímající“ a dále na novém řádku „tímto jsme dostali DKA, který je doplňkem automatu předchozího“. Za textem se poté už jenom objeví komponenta automatu s výsledným automatem.

6 Testování a nasazení serveru

Tato kapitola se týká testování serveru a jeho nasazení do provozu.

6.1 Testování

Testování je jedna z nejzávažnějších součástí vývoje aplikace. Tuto aplikaci jsem testoval rovnou během vývoje a na konci vývoje pro více případů vstupů. K testování docházelo i na straně vedoucího diplomové práce. Testování se poté dá rozdělit na tyto části:

- **Testování uživatelského rozhraní** : grafické rozhraní aplikace bylo testováno pro všechny možnosti vstupu automatu, jak pomocí grafu, tak pomocí přechodové tabulky nebo pomocí XML souboru. Důraz je kladen zvláště na zobrazení automatu a modelu automatu, který je hlavní v komunikaci mezi serverem a prohlížečem.
- **Testování pro různé prohlížeče** : v rámci splnění nefunkčních požadavků je aplikace otestována v následujících prohlížečích (v závorkách jsou verze prohlížečů)
  - GoogleChrome (57.0.2987)
  - Mozilla Firefox (44.0.2)
  - Opera (44.0)
  - Microsoft Edge (38.14393)
- **Testování jednotlivých algoritmů** : všechny algoritmy byly otestovány oproti různým příkladům automatu, čerpány jak z příkladů ve scriptech teoretické Informatiky, tak příkladům náhodně vymyšleným.

6.2 Nasazení

Pro testování je možné použít vestavěné IIS Express v prostředí Microsoft Visual Studio, nebo také rovnou plnohodnotné IIS jak už lokální, tak na nějakém serveru s IIS. Microsoft Visual Studio má možnost se na toto IIS připojit a nahrávat výslednou aplikaci už při vývoji. Pro nasazení do reálného provozu je třeba server s podporou technologie ASP.NET (nejlépe právě IIS od Microsoftu).

K nasazení pak stačí výsledný obsah složky, kde se projekt nachází, zkopírovat na server. Nebo se může předem projekt ještě zkompilovat pomocí publikace v Microsot Visual Studiu a následně výsledné soubory překopírovat na server.

7 Závěr

Hlavním cílem této diplomové práce bylo vytvoření serveru pro podporu výuky předmětu Teoretická informatika. Těchto diplomových prací je více a jsou rozděleny na oblasti, které se v tomto předmětu vyučují. Mým přesným úkolem bylo vytvořit server pro výuku algoritmů z oblasti konečných automatů.

Důležitou součástí práce bylo pochopení učiva zabývající se konečnými automaty. Základy těchto znalostí jsem získal v předmětu Úvod do teoretické informatiky ještě na bakalářském studiu na VŠB v Ostravě a poté jsem si tyto znalosti obohatil v předmětu Teoretická informatika také na VŠB. Dále jsem pro teorii v této práci využíval skripta Teoretická informatika [1] a Úvod do teoretické uformatiky [2].

Diplomovou práci jsem začal programovat ze začátku od komponenty automatu, pro vstup i výstup konečného automatu, nejdřív grafem a poté přechodovou tabulkou. A dále jsem pokračoval přímo algoritmy související s touto diplomovou prací.

Aplikace v konečné fázi umožňuje pochopit látku související s algoritmy používající konečné automaty, jak grafickým způsobem, tak i pomocí komentářů k jednotlivým krokům daného algoritmu. Dále umí výsledné automaty ukládat pro další nasazení v jiném nebo stejném algoritmu. Například po sjednocení nebo průniku dvou automatů se výsledný automat může uložit a poté načíst u algoritmu pro redukci automatu. Dále se na výsledný server můžou kdykoliv nahrát nové příklady pro studenty, při kterých nebudou muset automat ručně vkládat.

Celkovým přínosem této práce je snadnější výuka v oblasti konečných automatů. Pro mne osobně bylo přínosem seznámení a pochopení s prací na HTML5 canvasu a dále také vývoje pomocí technologie jako je ASP.NET WEB API 2. Procvičil jsem si také programování algoritmů, s velkou pomocí nástroje LINQ v .Net Frameworku, skriptování pomocí Javascriptu na straně prohlížeče s celkovým vytvářením dynamických HTML stránek.

Celý server je nyní připraven k nasazení do reálného provozu. Pro účely testování jsem jej zprovoznil na neplaceném serveru na adrese: `http://kativsb.aspone.cz/`.

Literatura

[1] Petr Jančar, CSc. Teoretická informatika. Ostrava 2007 [cit. 2017-04-28] *Dostupné z* `http://www.cs.vsb.cz/jancar/TEORET-INF/ti-text.2010-01-20.pdf`

[2] Petr Jančar, CSc. Úvod do teoretické informatiky. Ostrava 2007 [cit. 2017-04-28] *Dostupné z* `http://www.cs.vsb.cz/sawa/uti/materialy/uti.pdf`

[3] Understanding Detailed Architecture of ASP.NET 4.5, Dotnettricks [online], [cit. 2017-05-01] *Dostupné z* `http://www.dotnettricks.com/learn/aspnet/understanding-detailed-architecture-of-aspnet-45`

[4] Microsoft Developer Network [online], [cit. 2017-05-02] *Dosupné z* `https://msdn.microsoft.com/`

[5] Patrick Horgan, DBP Consulting [online], [cit. 2017-05-02] *Dostupné z* `http://dbp-consulting.com/`

[6] Dragging objects – a very simple HTML5 Canvas example, Rectangleworld [online], [cit. 2017-05-02] *Dostupné z* `http://rectangleworld.com/blog/archives/15`

Příloha A - příloha na CD

obsahuje diplomovou práci, dokumentaci a zdrojové kódy celé aplikace

- **dokumentace.xml** - programátorská dokumentace
- **jan0292.pdf** - text diplomové práce
- **publish/** - zkompilovaná aplikace pro nasazení na server
- **project/** - zdrojový kód celé aplikace